DESIGN AND ANALYSIS OF HASH FUNCTIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ONUR KOÇAK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
CRYPTOGRAPHY

JULY 2009

Approval of the thesis:

DESIGN AND ANALYSIS OF HASH FUNCTIONS

submitted by **ONUR KOÇAK** in partial fulfillment of the requirements for the degree of **Master of Science in Department of Cryptography, Middle East Technical University** by,

Prof. Dr. Ersan AKYILDIZ _____
Director, Graduate School of **Applied Mathematics**

Prof. Dr. Ferruh ÖZBUDAK _____
Head of Department, **Cryptography**

Assoc. Prof. Dr. Ali DOĞANAKSOY _____
Supervisor, **Department of Mathematics**

Examining Committee Members:

Prof. Dr. Ersan AKYILDIZ _____
Department of Mathematics, METU

Assoc. Prof. Dr. Ali DOĞANAKSOY _____
Department of Mathematics, METU

Dr. Muhiddin UĞUZ _____
Department of Mathematics, METU

Dr. Meltem Sönmez TURAN _____

Dr. Nurdan SARAN _____
Department of Computer Engineering, Çankaya University

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ONUR KOÇAK

Signature :

ABSTRACT

DESIGN AND ANALYSIS OF HASH FUNCTIONS

KOÇAK, Onur
M.S, Department of Cryptography
Supervisor : Assoc. Prof. Dr. Ali DOĞANAKSOY

July 2009, 63 pages

Hash functions are cryptographic tools that are used in various applications like digital signature, message integrity checking, password storage and random number generation. These cryptographic primitives were, first, constructed using modular arithmetical operations which were popular at that time because of public key cryptography. Later, in 1989, Merkle and Dåmgard independently proposed an iterative construction method. This method was easy to implement and had a security proof. $MD-4$ was the first hash function to be designed using Merkle-Damgård construction. $MD-5$ and SHA algorithms followed $MD-4$. The improvements in the construction methods accordingly resulted in improvements and variations of cryptanalytic methods. The series of attacks of Wang et al. on MD and SHA families threaten the security of these hash functions. Moreover, as the standard hashing algorithm $SHA-2$ has a similar structure with the mentioned hash functions, its security became questionable. Therefore, NIST announced a publicly available contest to select the new algorithm as the new hash standard $SHA-3$.

The design and analysis of hash functions became the most interesting topic of cryptography. A considerable number of algorithms had been designed for the competition. These algo-

rithms were tested against possible attacks and proposed to NIST. After this step, a worldwide interest started to check the security of the algorithms which will continue untill 4^{th} quarter of 2011 to contribute to the selection process.

This thesis presents two important aspects of hash functions: design and analysis. The design of hash functions are investigated under two subtopics which are compression functions and the construction methods. Compression functions are the core of the hashing algorithms and most of the effort is on the compression function when designing an algorithm. Moreover, for Merkle-Damgård hash functions, the security of the algorithm depends on the security of the compression function. Construction method is also an important design parameter which defines the strength of the algorithm. Construction method and compression function should be consistent with each other. On the other hand, when designing a hash function analysis is as important as choosing designing parameters. Using known attacks, possible weaknesses in the algorithm can be revealed and algorithm can be strengthened. Also, the security of a hash function can be examined using cryptanalytic methods. The analysis part of the thesis is consisting of various generic attacks that are selected to apply most of the hash functions. This part includes the attacks that NIST is expecting from new standard algorithm to resist.

Keywords: Hash Functions, Applications of Hash Functions, Design of Hash Functions, Analysis of Hash Functions

ÖZ

ÖZET FONKSİYONLARIN TASARIMI VE ANALİZİ

KOÇAK, Onur
Yüksek Lisans, Kriptografi Bölümü
Tez Yöneticisi : Doç. Dr. Ali DOĞANAKSOY

Temmuz 2009, 63 sayfa

Özet fonksiyonlar sayısal imza, veri bütünlüğünün kontrolü, şifre saklama ve rassal sayı üretimi gibi birçok uygulamada kullanılan kriptografik araçlardır. Özet fonksiyonlar önceleri açık anahtarlı şifrelemenin duyurulmasıyla kullanımı artan modüler aritmetik işlemlerinden yararlanılarak tasarlanıyordu. Daha sonra 1989'da bağımsız olarak Merkle ve Damgård tarafından yinelemeli bir tasarım metodu duyuruldu. Bu methodun uygulanması kolaydı ve bir güvenlik ispatı vardı. Tasarım metodlarındaki gelişmeler, analiz metodlarının da gelişip çeşitlenmesine sebep oldu. Wang ve ekibinin MD ve SHA ailesine yaptığı saldırılar $MD-5$ ile $SHA-0$ ve $SHA-1$ i tehdit ederken, aynı yapıdaki standart özet algoritması $SHA-2$ 'nin güvenliğini de şüpheye düşürüyordu. Bu sebeple NIST, daha güvenli bir özet fonksiyonu seçmek için, açık katılımlı bir yarışma düzenledi.

Özet fonskiyonların tasarımı ve analizi, bu yarışmanın duyurulmasıyla kriptografinin en ilgi çeken konusu haline geldi. Yarışma için birçok algoritma tasarlandı, güvenlikleri test edildi ve NIST'e gönderildi. Daha sonra da dünya çapında ilgililer tarafından güvenlikleri sınandı ve sınanmakta.

Bu tez özet fonksiyonların iki önemli yönünü sunmaktadır: tasarım ve analiz. Tasarım sıkıştır-

ma fonksiyonları ve yapı yöntemleri olmak üzere iki altbaşlık altında incelenmektedir. Sıkıştırma fonksiyonları algoritmanın çekirdeğini oluştururlar ve tasarım aşamasında en çok üzerinde durulan öğedir. Merkle-Damgård yapısındaki özet fonksiyonlarda algoritmanın güvenliği, sıkıştırma fonksiyonunun güvenliğine dayanır. Yapı yöntemi, algoritmanın güvenliğini belirleyen bir başka tasarım öğesidir. Yapı yöntemi ile sıkıştırma fonksiyonu birbirleriyle tutarlı olmalıdır. Diğer taraftan, tasarım aşamasında analizler de tasarım öğelerinin seçimi kadar önemlidir. Bilinen ataklar yardımıyla algoritmadaki zayıflıklar ortaya çıkartılıp algoritma güçlendirilebilir. Ayrıca kullanımda olan bir özet algoritmasının güvenliği de bu yöntemlerle sınanabilir. Tezin analiz bölümü birçok algoritmaya uygulanabilen ataklardan oluşmaktadır. Bu ataklar arasında NIST tarafından seçilecek algoritmanın dayanıklı olması özellikle vurgulanan ataklar da bulunmaktadır.

Anahtar Kelimeler: Özet Fonksiyonlar, Özet Fonksiyonların Uygulamaları, Özet Fonksiyonların Tasarımı, Özet Fonksiyonların Analizi

To my family...

ACKNOWLEDGMENTS

In the first place, I owe my deepest gratitude to my supervisor Ali DOĞANAKSOY for his support and guidance. Without his guidance, this thesis would barely have begun, much less been completed. He deserves another thank by introducing me to cryptography.

Special thanks go to Neşe ÖZTOP for her endless support and encouragement.

It is a great pleasure to thank Meltem Sönmez TURAN for her advices, corrections, constructive criticism and positive energy. I would like to mention my colleagues Barış EGE and Çağdaş ÇALIK for their suggestions and nice working atmosphere. It was a pleasure to work with them.

I cannot end up without thanking to my family whose constant support and love I have relied throughout all my life. I also want to thank my friends, everyone at IAM and other members of my family for believing in me.

Finally, I would like to acknowledge the financial support of the Scientific and Technological Research Council of Turkey (TUBITAK).

TABLE OF CONTENTS

ABSTRACT iv

ÖZ vi

DEDICATION viii

ACKNOWLEDGMENTS ix

TABLE OF CONTENTS x

LIST OF FIGURES xii

CHAPTERS

1 INTRODUCTION 1

1.1 Definitions 1

1.2 Properties of Hash Functions 2

1.3 Applications of Hash Functions 4

1.3.1 Digital Signature 5

1.3.2 Message Integrity 6

1.3.3 Authentication Protocols 7

1.3.4 Password Protection 7

1.3.5 Random Number Generation 8

1.4 Thesis outline 8

2 COMMON COMPRESSION METHODS 10

2.1 Block Cipher Based Hash Functions 10

2.2 Stream Cipher Based Hash Functions 13

2.3 Dedicated Hash Functions 14

2.4 Modular Arithmetic Based Hash Functions 15

2.5 Knapsack Based Hash Functions 16

2.6 Some Other Types of Hash Functions 17

3 CONSTRUCTION METHODS 18
3.1 Merkle-Damgård Construction 18
3.2 HAIFA Construction 20
3.3 Sponge Construction 21
3.4 Some Other Hash Constructions 22
3.4.1 Wide Pipe and Double Pipe Construction 22
3.4.2 Prefix-Free Merkle-Damgård Construction 23
3.4.3 Enveloped Merkle-Damgård Construction 23
3.4.4 RMX Construction 24
3.4.5 3C and 3C-X Constructions 24
3.4.6 Dynamic Hash Function Construction 25
4 ATTACKS on HASH FUNCTIONS 27
4.1 Birthday Attack 28
4.2 Correcting Block Attack 30
4.3 Meet in the Middle Attack 31
4.4 Length Extension Attack 33
4.5 Joux Attack 36
4.6 Fixed Point Attack 39
4.7 Long Message Attack 42
4.8 Kelsey & Schneier's Long Message Attack 43
4.9 Herding Attack 47
4.10 Slide Attack 51
4.11 Rebound Attack 54
5 CONCLUSION 58
REFERENCES 59

LIST OF FIGURES

FIGURES

Figure 1.1 Classification of Hash Functions 2

Figure 1.2 Digital Signing and Verification 5

Figure 1.3 Digital Signing and Verification Using Hash Functions 6

Figure 1.4 A Simple Authentication Protocol 7

Figure 2.1 Davies-Meyer, Matyas-Meyer-Oseas and Miyaguchi-Preneel constructions respectively. 11

Figure 2.2 MDC-2 Hash Function 11

Figure 2.3 MDC-4 Hash Function 12

Figure 2.4 Panama Hash function Push Mode 13

Figure 2.5 Panama Hash function Pull Mode 14

Figure 2.6 Step Function of MD5 15

Figure 3.1 MD Structure 19

Figure 3.2 Sponge Construction 22

Figure 3.3 EMD Construction 24

Figure 3.4 3C Construction 25

Figure 4.1 Model A 32

Figure 4.2 Model B 32

Figure 4.3 Model C 33

Figure 4.4 The Authentication Protocol 34

Figure 4.5 Expanding the Message 34

Figure 4.6 Output Transformation 35

Figure 4.7 Prefix Padding and Postfix Padding 35
Figure 4.8 Double Hashing 36
Figure 4.9 Multicollisions 37
Figure 4.10 Multicollisions for $m < h$ 37
Figure 4.11 Hashing Process of M 39
Figure 4.12 Hashing Process of M' 40
Figure 4.13 Coincidence in the Chaining Values h_k and h_l 40
Figure 4.14 Shrinking the Message 41
Figure 4.15 Davies-Meyer Construction 41
Figure 4.16 A $(k, k + 2^k - 1)$-Expandable Message 45
Figure 4.17 $(2, 5)-$Expandable Message 45
Figure 4.18 Diamond Structure 47
Figure 4.19 Sample Attack (m_{pad} is the padding block for guessed length) 49
Figure 4.20 Diamond Structure with an Expandable Message 50
Figure 4.21 Compression Function of Whirlpool Hash Function 55
Figure 4.22 Differential Characteristic for Rebound Attack on Whirlpool 55

CHAPTER 1

INTRODUCTION

Cryptographic hash functions have an important role in many applications. Digital signature, message integrity checking, authentication protocols, password protection and random number generation are some of them. Therefore, any flaw or weakness in a standard or popular hash algorithm affects various applications. For several years standard $SHA-1$, $SHA-2$ and non-standard $MD-5$ hashing algorithms are widely used. However, cryptanalytic attacks, especially attacks of Wang et al.[1, 2, 3, 4], and increasing computational power brings the need for a new and more secure hash function. For this reason, National Institute of Standards and Technology (NIST) announced a publicly available contest, similar to the AES contest, in which the winning algorithm will be selected as the new standard hashing algorithm $SHA-3$.

This contest makes hash functions the most interested topic of cryptography. First the design and later the analysis of hash functions became the hot topics of cryptographic community.

This thesis describes the methods of designing and analyzing hash functions. The main focus of this thesis is the unkeyed hash functions. Throughout this chapter, a brief introduction to hash functions will be given. Chapter 2 and Chapter 3 are devoted to designing blocks of hash functions. In Chapter 2 common compression functions and in Chapter 3 popular construction methods will be dealt with. Finally, in Chapter 4, attacks on hash functions will be mentioned.

1.1 Definitions

A cryptographic hash function is a mapping that takes arbitrary length input and gives a fixed size output. In practice, the input size is not arbitrary, but it is bounded by a very large number so that almost every possible data or message can be hashed at once. Output of the

hash function is called the *hash value*, *message digest*, or *digest value*. The size of hash value changes according to the algorithm and varies between 128 bits and 512 bits. A hash function which outputs $n - bit$ hash value is called an $n - bit\ hash$.

Hash functions can be classified according to their input parameters as shown Figure 1.1. Some hash functions take just one input parameter to produce a hash value while some take two parameters. Hash functions taking one parameter, message, to produce hash value, are called the *unkeyed hash functions* also named *Manipulation* (*or Modification*) *Detection Codes* (*MDC*) or *Message Integrity Codes* (*MIC*). As can be understood from the name(s), an unkeyed hash function concerns only the integrity of the message [5]. The other of type hash functions, taking two input parameters, message and the secret key, to generate a hash value, are called the *keyed hash functions* or *Message Authentication Codes* (*MAC*). MAC use a *key* to generate the hash value of a message. So, the generated hash value will contain information from the key besides the message. Therefore, MAC not only deals with the integrity of the message, but also the authenticity of the party who produced the message [5].

1.2 Properties of Hash Functions

The unkeyed hash functions further divided into two as *one way hash functions* (*OWHF*) and *collision resistant hash functions* (*CRHF*). Note that a hash function can be both one way and collision resistant which is in fact desired for cryptographic purposes.

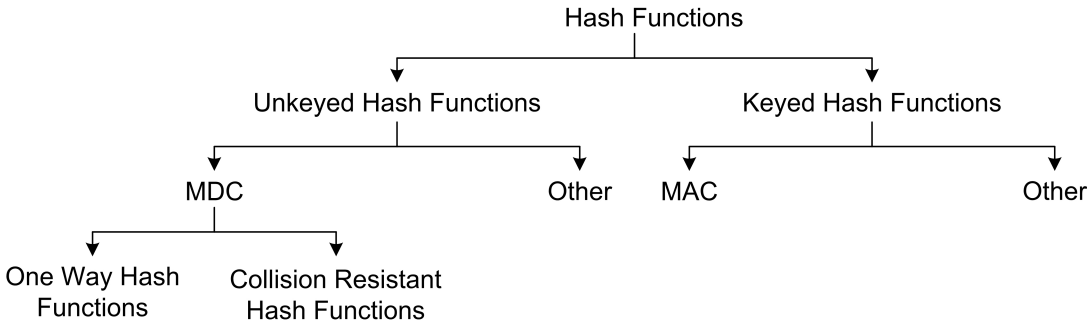

Figure 1.1: Classification of Hash Functions

One Way Hash Functions: A one way function f is basically a function which cannot be reversed in polynomial time i.e. given the output y of f it is infeasible to find x such that $f(x) = y$. With this notion, a one way hash function can be defined as a hash function H,

given the hash value $H(M) = y$, one cannot deduce the actual message M within a feasible time. More formally a function H is a one way hash function if it satisfies the following conditions

- Message M can be arbitrary length and the output should be fixed length.
- Given the output y of H, it is hard to find M st $H(M) = y$ (preimage resistancy) and given M_1, it is hard to find M_2 with $M_1 \neq M_2$, $H(M_1) = H(M_2)$ (second preimage resistancy).

The "hardness" or "infeasibility" is defined according to the bit-length of the hash value. For an n-bit hash function, a preimage or second preimage should not be found in less than 2^n hash function computations. For security reasons n should be greater than 160.

Collision Resistant Hash Function: A collision for a hash function H is the presence of two distinct messages M_1 and M_2 with $H(M_1) = H(M_2)$[1]. Since the input space of a hash function is greater than the output space the collisions are unavoidable. For a collision resistant hash function, finding colisions should not be trivial. More formally, a collision resistant hash function should satisfy the following conditions:

- Message M can be of arbitrary length and the output should be fixed length.
- The hash function should be one way, i.e., should be preimage resistant and second preimage resistant.
- It should be hard to find two distinct messages M_1 and M_2 with $H(M_1) = H(M_2)$.

Again the "hardness" of finding collisions depends on the size of the hash value. For a collision resistant n-bit hash function finding collisions should take at least $2^{n/2}$ hash function computations by birthday paradox (See section 4.1).

The Relation Between Properties

Some of the properties of hash functions imply the presence of other properties. Moreover, some properties seem to be related but in fact they can be unrelated. This section provides the relationship between properties.

[1] Note that there is no condition on the messages M_1 and M_2, while in the second preimage case, M_1 is a probabily given, fixed value.

If a hash function is collision resistant, then it is also second preimage resistant. Let H be collision resistant but not second preimage resistant. Then given a message M, finding another message M' such that $H(M) = H(M')$ is "easy" since H is not second preimage resistant. However, finding two distinct messages with $H(M) = H(M')$ with feasible complexity means finding a collision with feasible complexity and this contradicts with collision resistancy of H. Therefore H must be second preimage resistant.

On the other hand, collision resistancy does not imply preimage resistancy. For example, the function $f(x) = x$ is collision resistant but not preimage resistant. In order to achieve compression, consider the following example of [5] where $g(x)$ is a collision resistant n-bit hash function.

$$H(x) = \begin{cases} 0||x & \text{if } |x| = n \\ 1||g(x) & \text{else} \end{cases}$$

Here, for any $n + 1$-bit hash value with first bit '0', it is trivial to find the preimage x.

A hash function H may be preimage resistant but this does not guarantee that H is also second preimage resistant or collision resistant. One can define a hash function as $H(x) = x^2 \ mod \ n$ with $n = pq$ where p and q are very large prime numbers. In order to find the preimage, one should factorize n which is not feasible. However, x and $-x$ have the same hash value, therefore H is neither second preimage nor collision resistant.

Additional Properties: The properties mentioned above are necessary but they are not the only properties that a hash function should satisfy. Also non-correlation, near collision resistancy, partial preimage resistancy [5], complementation freedom, addition freedom and multiplication freedom [6] are some of the desired properties of a cryptographic hash function.

1.3 Applications of Hash Functions

Hash functions are used in various applications. In this section only a few of them will be mentioned.

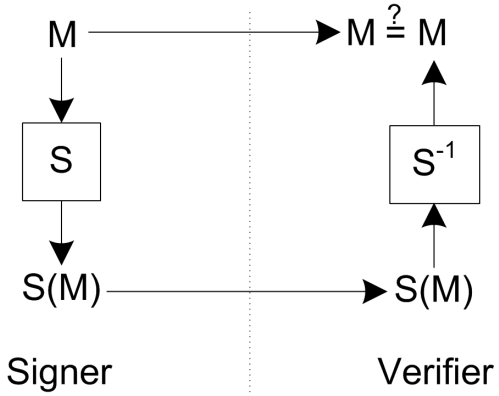

Figure 1.2: Digital Signing and Verification

1.3.1 Digital Signature

Digital signature is an asymmetric cryptographic application which is equivalent to handwritten signature in electronic media. Digital signing process can be seen as the reverse of the public key encryption process. When signing a message, the sender encrypts the message M with his secret key d and sends with the signature $S(M)$ with the message. The receiver decrypts the signature using the public key of the sender e and checks if the decrypted signature is equal to the received message as in Figure 1.2.

However, both signing a message and verifying its signature are costly processes. Most operations are taking powers of large numbers in a large module which requires considerable computation power especially for long messages. Moreover, the signature on a message is of the same size with the signed message. This doubles the required storage area and bandwidth.

Hash functions are integrated to the signing standards in order to reduce the cost of digital signature. Instead of the message, the hash value of the message is signed. This operation is more efficient than signing the message itself since hash functions are very fast compared to signing algorithm and signing algorithm is applied to a very short data. The verification is again on the hash values: the receiver hashes the received message and checks if the signature is on the hash value of the message or not. Figure 1.3 shows the signing and the verification processes.

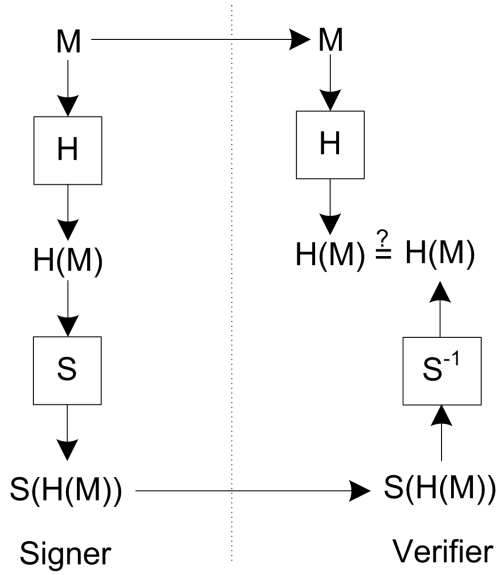

Figure 1.3: Digital Signing and Verification Using Hash Functions

1.3.2 Message Integrity

The most basic aim of hash functions is the integrity checking. By using hash functions one can detect if any change occurred in a data or not.

In order to check the integrity of a transmitted message sender hashes the message and sends both the message and the hash value. Hash value is sometimes sent from a secure channel, however, generally sent with the message from an insecure line. The receiver hashes the received message and checks the resulting hash value with the received hash value. If two hash values do not match, then it is clear that the integrity of the message is not preserved. If they match, message integrity is preserved. It is very unlikely that both message and hash value alter so that the altered hash value is the hash value of the altered message. The probability of such an event is 2^{-n} for an n-bit hash.

Also the integrity of a stored data can be verified by using hash functions. One can store the hash value of a data, probably at a different location, and when necessary can compare the stored hash value with the current hash value of the data. If the hash values match, the integrity of the data is preserved.

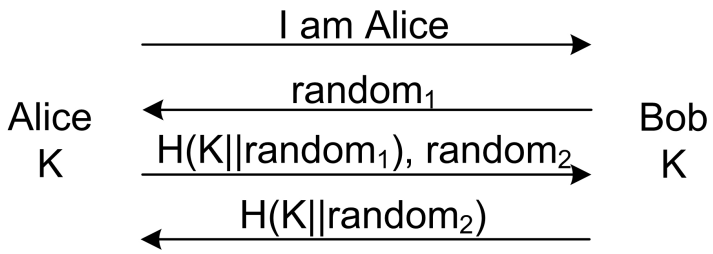

Figure 1.4: A Simple Authentication Protocol

1.3.3 Authentication Protocols

An authentication protocol is a cryptographic protocol in which the communicating parties authenticate themselves to the other party as in Figure 1.4. These protocols generally rely on a shared secret like a password. Therefore, in these communications the transmission should be encrypted. However, if public key encryption is used, the protocol can be very slow, or if a symmetric key algorithm is used another shared key will be necessary. Also using the same key for a symmetric encryption so many times is a security weakness. So hash algorithms are used in these protocols, generally with public key encryption. The shared secret and reply to the challenge of the authenticating party is hashed and encrypted before sent by the sender. This is also a more secure way to communicate since if an attacker can forge the encryption schema and get the plaintext, she cannot derive the secret. Moreover since hashing algorithms are generally faster than the encryption algorithms, using hash functions is more efficient.

1.3.4 Password Protection

Password storage is another daily life application where hash functions are used intensively. Nearly all local or remote systems use an authentication protocol to authenticate the users. In most of the cases the user enters his user name and password and the other party checks this password if it is the password of the claimed user. Therefore, the user name of the user and his password should be, somehow, stored in the authenticating part.

Storing the user names and passwords in cleartext is not a good idea since anyone who can access this file will get the passwords of all the users of the system. Encryption is a possible solution, however, the system administrators can have the knowledge of the encryption key, and can access to the passwords. Therefore, these files include the user name and the hash

value of the password for that user instead of the password itself. This way accessing the file will not be enough to get the passwords of the users, a preimage or second preimage for the hashing algorithm will be needed. An extra encryption on this file can also be applied to increase the security. In the authentication process, user enters his user name and password. The authenticating party hashes the password and compares with the stored hash value for the entered user name. In most of the password storage applications, password is hashed with a random value salt. This way, users with the same password will have distinct corresponding hash values.

1.3.5 Random Number Generation

Random numbers are important primitives for cryptography and applications where it is necessary to produce unpredictable results. In public key cryptography, most of the time one of the prime multipliers of the modulo n is generated randomly. Besides, keys in the symmetric encryption schemes should be random.

There are a lot of pseudo random number generator algorithms. However, these algorithms generate random numbers by using an input called seed and the generated numbers are related with this seed. Therefore, the seed should also be random or it should be a function of seeds from different sources.

A good hash function should not be distinguished from a random oracle which means the output of the hash function is random. Moreover, hash functions are fast algorithms. Therefore they are good candidates for random number generators.

1.4 Thesis outline

This thesis studies the design and analysis of hash functions. Most common and secure designing methods, and attacks that can be applied to many hash functions are presented.

Chapter 1 is an introduction for the cryptographic hash functions, their properties and applications.

Chapter 2 is devoted to the compression functions which are cores of the hash functions. This

chapter includes all common compression functions.

Chapter 3 is also on the design of hash functions. In this chapter, construction methods are examined in details. Besides the most common construction method, Merkle-Damgård, variants of this construction and a new construction method *sponge construction* are reviewed.

Chapter 4 is a comprehensive chapter on the attacks against hash functions. This chapter includes generic attacks and the precautions against these attacks.

CHAPTER 2

COMMON COMPRESSION METHODS

Compressing the data is the first objective of hash functions. A hash function should compress the input so that the resulting hash value could be used as a fingerprint of the input which can stand for the input itself for many applications.

There are several types of hash functions that use different methods to compress the data. Some of these methods depend on the existing cryptographic tools while others are derived from mathematical problems.

In this chapter, the most common types of hash functions are investigated in detail. The most widely used method is using a block cipher as a compression function. However, recently the number of stream cipher based hash functions is increasing. Modular arithmetic and knapsack problem are other tools that are used in hash functions for compression. Apart from these constructions, there are some other methods which are not very common.

2.1 Block Cipher Based Hash Functions

Block cipher based hash functions use block ciphers as compression functions. The main idea of using a block cipher as a compression function is the minimization of designing efforts and the implementation costs. A hash function should not be distinguished from a random oracle. Therefore, using a secure block cipher is preferable than designing a new compression function which behaves like a random oracle from scratch. Besides, if encryption and hashing operations are used together, using the same or similar algorithms for both encryption and hashing will reduce the implementation cost.

The first block cipher based hash functions were using DES as the compression function. After proposal of provably secure Merkle-Damgård construction, these hash functions became more popular. Most of the popular hash functions, like MD4, MD5, SHA-1 and SHA-2, use block ciphers as their compression functions.

Since the input size of a block cipher is relatively small than the message, message is input to the cipher in blocks. Each block is used for some number of encryptions. The number of message blocks per encryption is called the rate of the hash function. Some hash functions have low rate to be more efficient while some hash functions have higher rates for higher security. The constructions Davies-Meyer[1], Matyas-Meyer-Oseas [7], Miyaguchi-Preneel [8, 9] have rate 1, MDC-2 [10] has rate 1/2 and MDC-4 [10] has rate 1/4.

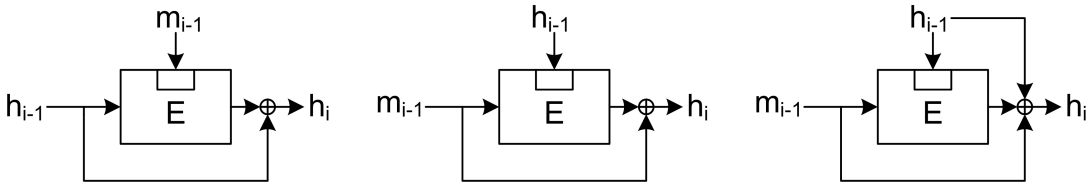

Figure 2.1: Davies-Meyer, Matyas-Meyer-Oseas and Miyaguchi-Preneel constructions respectively.

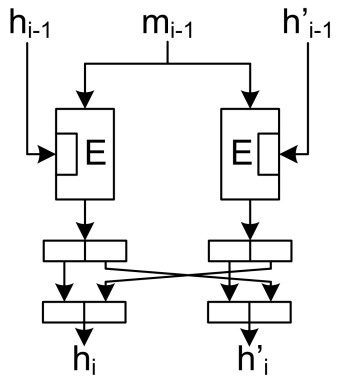

Figure 2.2: MDC-2 Hash Function

Block cipher based hash functions can be classified according to their output sizes as single block length hash functions, double block length hash functions and multi block length hash functions. Most block cipher based hash functions are designed on single or double block length approach.

[1] This construction is devoted to Meyer by Davies [5]

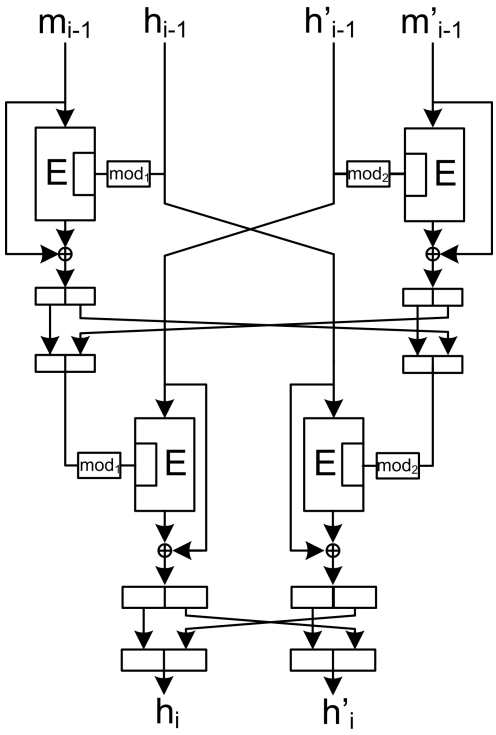

Figure 2.3: MDC-4 Hash Function

The compression functions of single block length hash functions output chaining values which have size equal to the ciphertext size of the block cipher. Since block ciphers are bijective, there should be additional operations for the inputs of the compression function to destroy invertability. The inputs of a block cipher are the plaintext P and the key K and these two inputs can be both chosen from the set $T = \{m_i, h_i, m_i \oplus h_i, c\}$ where m_i is the message block, h_i is the chaining value and c is constant. Moreover, there can be a feed forward operation FF with an element to satisfy the preimage resistency which can again be chosen from the set T. Therefore, there are totally 64 possible constructions. Preneel, Gavaerts and Vanderwale (PGV), stated that 12 of the constructions are secure by checking all the 64 designs [9]. Later, Black et al. [11] verified the collision resistance of 12 schemes proposed to be secure by PGV and also found that there are 8 more constructions which are collision resistant when iterated properly. In [12], Stam showed the security of these 20 constructions by deriving the secure schemes.

Since the ciphertext size of most block ciphers is too short to be a chaining value, some block cipher based hash functions doubles the output size of the compression function. The compression functions of double block length hash functions use two parallel encryption op-

erations and output a chaining value of double length of the ciphertext. MDC-2 and MDC-4 are double block length hash functions.

Later, Preneel and Knudsen [13] defined block cipher based hash functions with chaining values longer than double block length by using quaternary codes.

2.2 Stream Cipher Based Hash Functions

Block cipher based hash functions have restrictions on output sizes. To satisfy the security criterias at least double block length hash functions should be used. However, this decreases the speed and efficiency of the hash function.

Recently, hash functions are constructed based on stream ciphers for speed and efficiency. These hash functions can be constructed in two ways: as in block cipher case a stream cipher can be used as compression function or the hash algorithm can be designed as a stream cipher which takes message in blocks, mixes it and gives hash value like a key stream. Nearly all of the stream based hash functions are designed using the second approach.

The first stream cipher based hash function Panama was proposed in 1998 [14]. Panama was designed for both streaming and hashing purposes. The algorithm is based on a buffer b which acts like a linear feedback shift register, a state a and some transformatins ρ. The hash mode of Panama has three phases. First phase is the *push* mode where the data is input to the algorithm as in Figure 2.4. When all the data is used up the hash function is clocked without any input or output for 32 rounds to mix the message. In this phase, *blank pull*, state feeds the buffer instead of message blocks. Then the algorithm is clocked as many times as needed, with 256 bits output at each clock, to produce the hash value in *pull* phase as Figure 2.5.

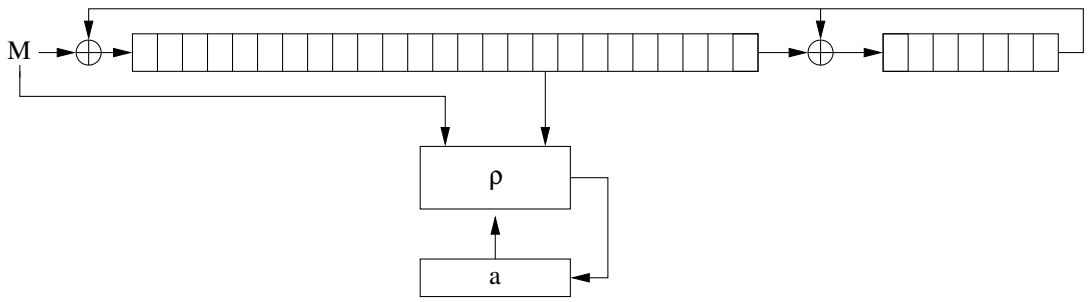

Figure 2.4: Panama Hash function Push Mode

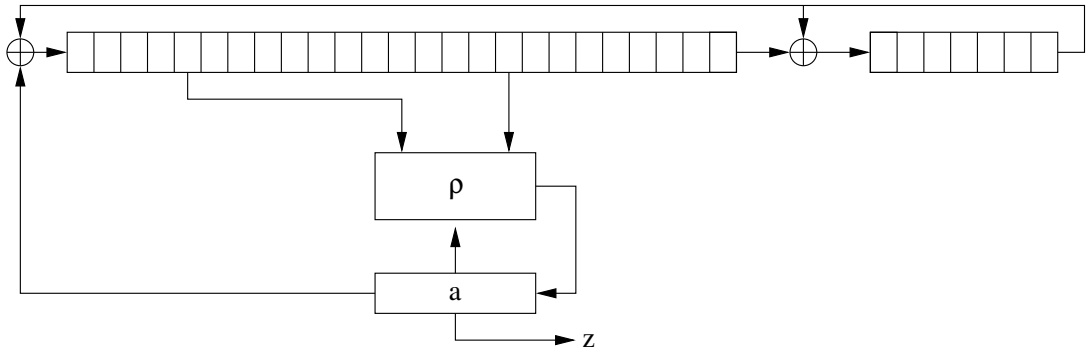

Figure 2.5: Panama Hash function Pull Mode

After Panama, RC4-Hash [15] was designed based on the stream cipher RC4. Later, after the attacks [16, 17] that complately broke Panama, designers proposed Radiogatun [18] in which the weaknesses of Panama are fixed. Moreover, there are some stream cipher based candidate algorithms for the NIST SHA-3 Contest like Lux [19] and Boole [20]. Stream cipher based hash functions are fast and efficient compared to block cipher based designs and hash functions based on mathematical problems. However, there is no security proof for these hash functions. Therefore to achive the claimed security some additional tools should be used.

In addition, stream cipher based hash functions can be viewed as iterated hash functions in the following manner: at each iteration a message block m_i and a state S_i is input to the algorithm and a state S_{i+1} is output which is the state obtained by clocking S_i with m_i.

2.3 Dedicated Hash Functions

Constructing a hash function is a hard task. Designing a compression function is the most crucial part of this job. To reduce the construction efforts, some hash functions use existing cryptographic ciphers, especially block ciphers, as their compression function. However, most of the time, such hash functions does not have the desired properties on **security-speed-hardware cost** combination. Hash functions which are designed to have optimal combination on desired output sizes are called *dedicated hash functions*.

In general, dedicated hash functions are constructed upon block cipher based approach. The block ciphers used in dedicated hash functions are designed to have good hashing properties instead of good encryption properties with low implementation cost plus high speed and

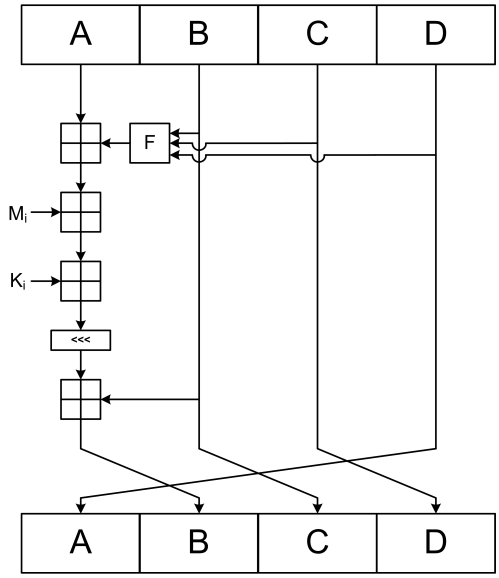

Figure 2.6: Step Function of MD5

security.

The first dedicated hash function to collect great attention is $MD-2$ [21] which is designed by Rivest in 1990. After collisions for this algorithm were found, same year Rivest proposed $MD-4$ hash function [22]. $MD-4$ design inspired many following hash functions. These functions are called the $MD\ family$. One year later, due to security concerns and Dobbertin's attack [23, 24], Rivest proposed a strengthened version of $MD-4$, namely $MD-5$ [25]. Then, NIST published a series of standards named $SHA\ family$ [26, 27], (*Secure Hash Algorithm*) which are variants of $MD-4$. Another improved version of $MD-4$ developed in RIPE project and resulted in the $RIPEMD\ family$.

2.4 Modular Arithmetic Based Hash Functions

Hash functions based on modular arithmetic reduce the security of the algorithm to difficulty of solving number theory problems. These hash functions can be used with public key cryptosystems to decrease the implementation efforts. Moreover, by changing the modulus used in the hash function, it is easy to scale the performance and security. However, these algorithms are very slow even when compared to block cipher based hash functions. Also fixed points, multiplicative attacks and attacks with small numbers like 0 and 1 are threatening for

the security of these algorithms.

Two of the important problems in number theory that can be used in hash functions are factorization problem and discrete logarithm problem.

One of the first modular arithmetic based hash function was an iterative hash function using ciphertext block chaining with RSA. Later some other hash functions designed following this approach [28, 29, 30, 31, 32, 33, 34, 35]. The simplest example of hash functions based on discrete logarithm problem is proposed by Shamir[2] as $H(M) = g^M \ mod \ N$ where $N = p \cdot q$ with p and q are large prime numbers and g is an element of maximum order in Z^n. Another RSA like hash algorithm, MASH-1(**M**odular **A**rithmetic **S**ecure **H**ash), repaired and redesigned a few times after successfull attacks, became an ISO standard in 1995 with MASH-2, a variant of MASH-1. These algorithms are still secure. The best known attacks for these hash functions require $2^{n/2}$ and 2^n complexity for collision and second preimage respectively.

2.5 Knapsack Based Hash Functions

The knapsack problem is an NP-complete problem that can be used in hashing. The problem can be defined as follows: Given a set of n integers $U = \{u_1, u_2, \ldots, u_n\}$, a modulus $l(n)$ and an integer z, does there exist a subset U' of U such that $\sum_{u_i \in U'} u_i = z \ mod \ d$.

The hash functions depending on the knapsack problem can be divided into two groups according to the knapsack they use: additive knapsack based hash functions and multiplicative knapsack based hash functions. Several additive knapsack based hash functions proposed and broken. Damgård designed a hash function depending on additive knapsack [33] and it was broken by Camion and Patarin [36, 37]. Later Ajtai [38] proposed a hash function and Goldreich et. al. [39] verified the collision resistancy of this function.

Besides additive knapsack based hash functions, there are also hash functions that uses multiplicative knapsacks. The first of these hash functions was announced by Bosset and broken ten years later by Camion again. Later two designs by Zemor [40] and Tillich and Zemor [41] are proposed. Attacks for the specific parameters of these proposals can be found in [42, 43].

Although knapsack problem is NP hard, lattice reduction technique can be used to attack these

[2] There is no proposal of the algorithm but reported by R.Rivest in a mailing list: http://diswww.mit.edu/bloom-picayune/crypto/13190

hash functions especially when $n \gg d$ [44].

2.6 Some Other Types of Hash Functions

In addition to the construction methods mentioned above there are other hash function classes which are very rare. These classes are hash functions based on cellular automata and hash functions based on matrix multiplication.

A simple cellular automata consists of a line of cells, where each cell can have a value 0 or 1. These cells are updated according to a rule at each step. There are two hash functions based on cellular automata which are Cellhash [45] and Subhash [46]. These hash functions are light, fast and until 2006 there were no attacks. In 2006 Chang [47] proposed a preimage attack for these hash functions.

Matrix multiplication is another tool that can be used in hashing algorithms. Banieqbal and Hilditch proposed Random Matrix Hashing Algorithm [48] which takes inputs as a $1 \times m$ row bits, multiplies with an $m \times n$ secret matrix and outputs $n \times 1$ column vector of bits. Another schema is invented by Harari [49]. In this schema, an $m \times m$ matrix is constructed from the message blocks and using a secret $t \times t$ matrix N the output is produced as $N^T \cdot M \cdot N$. However, chosen message attacks are applicable to this schema.

CHAPTER 3

CONSTRUCTION METHODS

In this chapter, construction methods of hash functions are discussed in details. Iterative hash functions still pace a big portion of all hash constructions. Therefore, iterative designs will be investigated deeply. These constructions include Merkle-Damgård(MD)[1], HAIFA and some other alternative designs derived from Merkle-Damgård. Besides, sponge constructions is included in this chapter. Sponge construction is a new and secure construction method which is becoming more popular. This thesis proposes the most common designs, their improvements and new designs that can be improved. Therefore, some constructions are not presented in this chapter. For example, for stream cipher based hash functions there are not many choices for the designer in construction since the hash function should be designed as a streaming module and the primitives or design rationales are specific to the algorithm. Also, hash functions based on mathematical problems considers the message as a number and the message is input according to the modulo used in the algorithm. Moreover, as the mentioned hash functions are not so wide they will not be mentioned in the following sections.

3.1 Merkle-Damgård Construction

Merkle-Damgård construction is the most widely used hash construction method, which was designed by R.Merkle [50] and I.Damgård [33] independently in 1989. Most of the hash functions and all the standardized hash functions are build upon MD construction.

MD construction is basically processed in three steps. First step is the padding step. The aim of the padding is to make the message length a multiple of message block length, m. The most

[1] To mention Merkle-Damgård construction, the abbreviation MD will be used while for hash functions from *MD* family the abbreviation will be italic and will be followed by an $-x$ indicating the version as in $MD-4$.

widely used padding procedure is as follows: a '1' bit followed by a number of '0' bits and the bitwise notation of the message length are appended to the message. The number of '0' bits appended to the message are chosen so that the length of the message becomes a multiple of block length m. In general the maximum length of the message that can be processed by the hash function is $2^{64} - 1$. Therefore 64 bit space is provided for the length padding. Considering the appended '1' bit, at least 65 bits are appended to all messages regardless of the message length. If l is the length of the message, d, the number of '0' bits appended is the smallest positive root of the equation

$$l + 65 + d \equiv 0 \ (mod\ m).$$

Second step is dividing the padded message into m bit blocks $m_0 m_1 m_2 \ldots m_{t-1}$. After this step, the chaining values are iteratively found by using a fixed, publicly known initialization vector, IV, and the message blocks:

$$\begin{aligned} h_0 &= IV \\ h_i &= f(h_{i-1}, m_{i-1}) \ \ i = 1, 2 \ldots, t \end{aligned}$$

where f is the compression function of the hashing algorithm.

Generally h_t is taken to be the hash value of the message. However an optional transformation $g(x)$ can be applied to h_t to get the hash value as $H(M) = g(h_t)$.

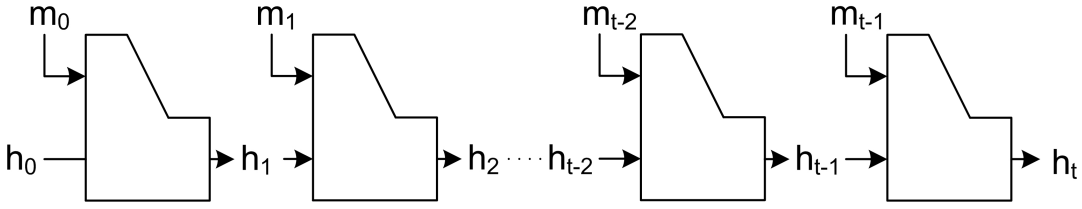

Figure 3.1: MD Structure

The iterative structure of MD construction enables arbitrary length messages to be processed easily by the hash functions. Also padding the message length and using a non-zero IV, which is called is called the $MD - strengthening$, increases the defeats or increases the complexity of various attacks.

The most important property of MD structure is that the collision resistance property of the compression function is preserved [50, 33]. The MD structure has a security proof that states

if the compression function used in the hash algorithm is collision resistant then the hash function itself is collision resistant [51].

Besides collision resistancy, it was believed that MD construction also preserves the preimage resistancy and second preimage resistancy of the compression function [52]. However there are several attacks in recent years against the second preimage resistency of the MD construction such as [53, 54, 55].

3.2 HAIFA Construction

MD structure is a provable collision resistant hash construction method. However, as computing power increases and new cryptanalytic tools are proposed, MD hash functions become weaker and more vulnerable to attacks. Biham and Dunkelman [52], fixing the flaws in MD structure, announced a new construction method HAIFA(**HA**sh **I**terative **F**r**A**mework). HAIFA is an iterative structure which is based on Merkle-Damgård and uses additional tools to increase the security.

The compression function f takes only message blocks m_i and chaining values h_i as inputs in MD hash functions. In HAIFA, compression function inputs are number of bits (or message blocks) hashed so far, b, and salt, s, in addition to message block and chaining value. So the chaining values can be expressed as $h_i = f(h_{i-1}, m_{i-1}, b, s)$.

The number of bits, or message blocks, hashed so far is included as an input to prevent the fixed point attacks. In MD structure attacker can freely inject fixed points to find a second preimage as she needs. However, even if b is not mixed well in f, since the inputs to the compression functions are different at each iteration, attacker can use a fixed point very limited times.

The other input, salt, is a precaution against attacks which has a precomputation phase. Some attacks have two phases as on-line and off-line phases. In the off-line phase attacker produces a number of structures which can be messages or chaining values using some weaknesses in the hash algorithm. After learning the pair $(M, H(M))$ she mounts the on line phase of the attack to produce a collision or second preimage. In HAIFA, attacker should know the salt to produce the structures but the salt is a random number which is sent with the pair $(M, H(M))$

as $(M, Salt, H(M, Salt))$ to the receiver. For security reasons the salt should be at least half length of the chaining value and should be random [52].

Moreover, in the HAIFA construction, there is a standardization of initial values for variable hash sizes. If a fixed IV is used for all hash sizes and if for some hash sizes hash values are truncated, then some bits of the hash values of distinct sizes will be equal. For example, if one needs a 224-bit hash value using a 256-bit hash function, he will produce the 256-bit hash value and truncate the final (or some other) 32-bits. Therefore, the 224 bits of the 256-bit and 224-bit hash values of a message will be common. HAIFA, uses different IVs for different hash values, so the truncation will not cause any problems.

Another addition is padding the hash size. Usual padding adds a '1' bit, some '0' bits to make the padded message a multiple of message block length m bits and the bitwise notation of the length of the original message. New construction method concatenates the bitwise notation of hash size after length padding. This ensures that no two messages exist which have the same hash value without truncation for different hash sizes.

3.3 Sponge Construction

Sponge construction [56] is a new construction method for hash functions and stream ciphers. This construction can be built upon a function f which can be expressed as a random permutation or random function. If f is expressed as a random permutation, construction is called a $P - sponge$, otherwise, if it is expressed as a random function, construction is called a $T - sponge$. The main difference between the compression functions of MD structures and the f function in sponge construction is, unlike the compressing functions in MD or Haifa, f is a function which maps l bit input to l bit output.

The construction is consisting of 2 phases: *absorbing* and *squeezing*. In the first phase, data is input to the sponge iteratively block by block and in the second phase output is given in the same manner. These two phases have similar procedures. In the absorbing phase, iteratively, message block is XOR-ed or overwritten to the state and f is applied to this state. Then, next message block is processed and so on. After all the message blocks are input, second phase is applied. In the squeezing phase, some part of the state is output and f is applied to the state. Then, again some part of the state is output and f is applied until the desired hash size

is achieved. This process is depicted in figure 3.2. Optionally, between two phases, some number of blank rounds can be applied. In the blank rounds, there is no input to or output from the state. Only, f is applied to the state.

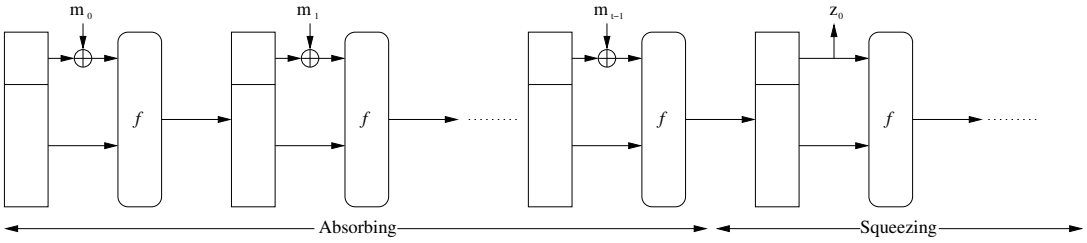

Figure 3.2: Sponge Construction

The size of the inner part of the state of sponge construction is called the *capacity* of the sponge function and denoted as c. The security of a sponge construction depends on its capacity c, hash size n and f function. For P-sponges, the complexity of a collision is $min(2^{c/2}, 2^{n/2})$, and complexity of preimage and second preimage is $min(2^{c/2}, 2^n)$. Collision complexity of T-sponges is equal to the collision complexity of P-collisions. Finding a preimage costs $min(2^c, 2^n)$, and finding a second preimage costs $min(\frac{2^c}{L}, 2^n)$ for a T-sponge, where L is the length of the original message.

This construction ensures that if f is a random function and $c \geq 2n$, then the sponge construction is indifferentiable from a random oracle.

3.4 Some Other Hash Constructions

3.4.1 Wide Pipe and Double Pipe Construction

In [57] and [58], Lucks introduced a new concept: wide pipe hash functions. These hash functions are constructed so that the intermediate chaining values are w bits for an n-bit hash size with $w > n$. There is a final transformation at the end which reduces the w-bit final chaining value to n bits.

The aim of the wide pipe construction is to increase the complexities of the attacks depending on chaining values. For example, for an $n-$bit hash H_1 with n-bit chaining values, an attack which finds collisions for the compression function with complexity $2^{n/4}$, means a break for

H_1. However for an n-bit wide pipe hash H_2 with w-bit chaining values where $w = 2n$, cost of this attack is same with exhaustive search cost and not a serious case as H_1.

Another approach, by Lucks, to widen the internal state is the double pipe hash functions. In this approach two parallel iterations are processed. These two iterations can be initialized with different IVs, can use different compression functions or can iterate the message blocks in different permutations. At the final step the outputs of the two iterations are mixed to get the hash value.

3.4.2 Prefix-Free Merkle-Damgård Construction

One of the problems of MD structure is about the randomness: although the underlying compression function is indifferentiable from a random oracle, the hash function itself is not guaranteed to be indifferentiable from a random oracle [59]. To solve this problem prefix-free encoding is suggested for iterated hash functions. Prefix-free encoding (or prefix encoding) is applied to the message before padding process. The two of the suggested encoding functions are

- $g_1(M) = L_M||m_0||m_1||\cdots||m_{t-1}$ where L_M is the message length.
- $g_2(M) = 0||\overline{m_0}||0||\overline{m_1}||0||\cdots||0||\overline{m_{t-2}}||1||\overline{m_{t-1}}$ where 0 and 1 are single bits and $\overline{m_i}$ is the $(i+1)^{th}$ $m-1$ bit block of the message M.

After encoding, the message is hashed as in MD structure and hash function becomes indifferentiable from a random oracle.

3.4.3 Enveloped Merkle-Damgård Construction

EMD is proposed by Bellare and Ristenpart in [60] which is a construction that preserves collision resistancy, preimage resistancy and pseudo-randomness of the compression function. The message blocks are iterated as MD up to the final message block. The final message block m_{t-1} and h_{t-1} are concatenated and taken as input to another iteration with a distinct IV. This operation is called *enveloping*.

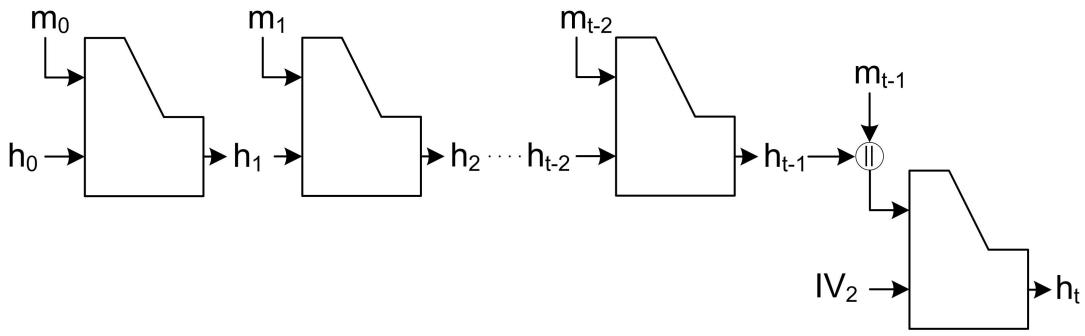

Figure 3.3: EMD Construction

3.4.4 RMX Construction

RMX is presented in [61] and [62] by Halevi and Krawczyk. The idea of RMX is randomization of the message before padding. In this method a random string, r, of length between smallest number of padding bits and message block length m is produced. From this random string r, three other parameters, r_0, r_1 and r_2, are produced: respectively by appending zero bits to r, repeating r as many times as necessary and taking some part of r. r_0 is prepended to the message, r_1 is XORed with all the message blocks except padding and r_2 is XORed with the padding blocks. After obtaining the hash value, r is stored, or send to a receiver, with the hash value.

$$\begin{aligned} h_1 &= f(IV, r_0) \\ &\vdots \\ h_i &= f(h_{i-1}, r_1 \oplus m_{i-2}) \; for \; i = 2, 3, ..., t \\ &\vdots \\ h_{t+1} &= f(h_t, r_2 \oplus m_{t-1}) \\ H(M) &= h_{t+1} \end{aligned}$$

3.4.5 3C and 3C-X Constructions

$3C$ and $3C - X$ are proposed by Gauravaram [63] which are similar to double pipe hash construction. In $3C$ construction, while message is iterated from the main line, there is an additional line which takes inputs from the main iteration line. Finally the outputs of two

lines are mixed to get the hash value. The algorithm can be defined as follows

$$
\begin{aligned}
h_1 &= f(h_0, m_0) \\
z_0 &= h_1 \\
h_i &= f(h_{i-1}, m_{i-1})\ i = 1, 2, \ldots, t \\
z_{i-1} &= f(z_{i-2}, PAD(h_i))\ i = 2, \ldots, t \\
h_{t+1} &= f(h_t, PAD(z_{t-1}) \\
H(M) &= h_{t+1}
\end{aligned}
$$

where $PAD(x)$ is padding x with 0 bits until it becomes m-bits.

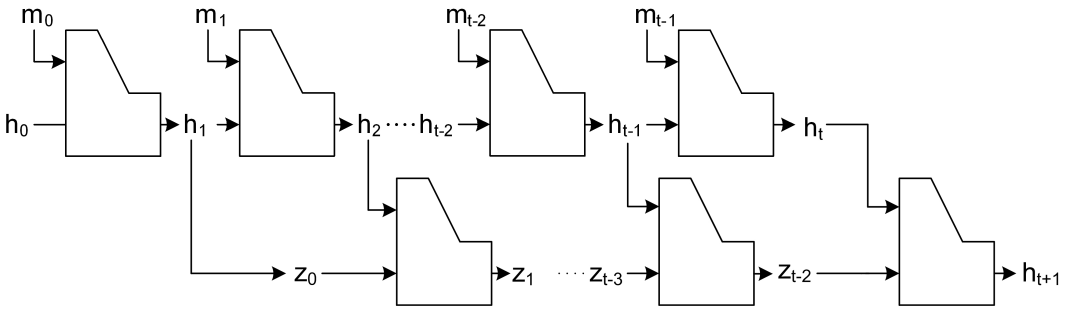

Figure 3.4: 3C Construction

The $3C-X$ construction is similar to $3C$. The only difference is that the compression functions in the second iteration line, except the final compression function, are replaced with XOR operations. This way construction becomes lighter.

3.4.6 Dynamic Hash Function Construction

Dynamic hashing, by Speirs [64], is a collection of iteration lines where each line feeds the next line. This construction has a security parameter s which can be considered as salt.

The message M is padded to be a multiple of $m - n$ bits. A '1' bit followed by a number of '0' bits are appended to the message. Then 64 bits message length and 32 bits representation of s is concatenated.

The number of lines, l, in the construction depends on hash size d. l is chosen such that $n(l-1) < d \leq nl$. Each line has an initial value, h_0^i which is derived from the line number

and s as $h_0^i = f(IV_1, IV_1||s||00..0||i-1)$. The lines are interacting with each other. The output h_k^j of k^{th} iteration of line j is concatenated to m_k and input to the $(k+1)^{st}$ iteration of line $(j+1)\ mod\ l$, where l is the number of lines.

At the end of iterations an "envelope" operation is done with another initial value $h_{t+1}^i = f(IV_2, h_t^i||00\cdots0)$ and these chaining values are concatenated to get the hash value using the iterations

$$
\begin{aligned}
IV_{s,j} &= f(IV_1, IV_1||s||00\cdots0||j-1)\ for\ j = 0, 1, \ldots, l-1, \\
h_{1,j} &= f(IV_{s,j}, m_0||IV_{s,(j-1)\ mod\ l})\ for\ j = 0, 1, \ldots, l-1, \\
&\vdots \\
h_{i,j} &= f(h_{i-1,j,m_{i-1}||h_{i-1},(j-1)\ mod\ l})\ for\ i = 2, \ldots, t-1,\ j = 0, 1, \ldots, l-1, \\
&\vdots \\
h_{t,j} &= f(IV_2, h_{t-1,j}||j)\ for\ j = 0, 1, \ldots, l-1, \\
H(M) &= g(h_{t,0}||h_{t,1}||\cdots||h_{t,l-1},)
\end{aligned}
$$

where $g(x)$ is a mapping from $n \cdot l$ bits to hash size.

CHAPTER 4

ATTACKS on HASH FUNCTIONS

Hash functions are used in various applications which require specific properties. Therefore, these cryptographic tools are designed in order to satisfy the desired properties. However, designing a hash function is not trivial and sometimes processes used in the algorithm may have some weaknesses, that are not obvious in the design-time, or interact with each other and result in a flaw. So, analysis of hash functions are crucial for the security of applications.

The security analysis of hash functions are mainly focused on finding preimage, second preimage and collisions. If, any of preimage, second preimage or collisions can be found with an effort less than $2^n, 2^n$ and $2^{\frac{n}{2}}$ respectively, then hash function is considered to be insecure against that kind of attacks or broken. Moreover, besides mentioned attacks, there are also *pseudo* and *near* attacks. In pseudo attacks attacker tries to find preimage, second preimage or collisions using different initial values for each message. In near attacks, she tries to find some part of the message or hash value instead of fully recovering them. These types of attacks are not so severe and will not be mentioned in this thesis.

The methods for analyzing hash functions, or attacks shortly, can be classified in various types according to the main parameters used in the attack. For example some attacks depend only on the hash size while other attacks may also depend on chaining value or compression function. At this point, one can categorize attacks, in a more general fashion, as generic attacks and specific attacks. Generic attacks are general attacks that are mostly applicable to numerous hash functions. Specific attacks, however, are cryptanalysis methods for specific hash functions and applicable to very limited number of algorithms. Specific attacks are out of the scope of this thesis and generic attacks will be mentioned in this section.

Before going into details of the attacks, a table covering the aims and the assumptions of the

attacks is given in Table 4.

Attack	Given	Aim
Preimage	$H(M)$	Find M
Second Preimage	M & $H(M)$	Find $M' \neq M$ with $H(M') = H(M)$
Collision	-	Find $M' \neq M$ with $H(M') = H(M)$

4.1 Birthday Attack

Birthday attack is the basic tool to find collisions for any hash function. The attack is based on the birthday problem (or birthday paradox) which is about the minimum number of people so that the probability of at least two people having the same birthday is greater than $1/2$. To bind the problem to hash functions, one can call the presence of two people having the same birthday a collision.

The key point in birthday attack is that the attacker, is looking for any collision rather than a specific collision. If attacker was looking for a specific collision among N people, say at least one person having the same birthday with her, the probability will be

$$P_{Collision}(N) = 1 - \left(\frac{364}{365}\right)^N.$$

For this probability being greater than $1/2$, N should be greater than 254. Applying this idea to hash functions, attacker is trying to find a message which has a hash value with a specific message by exhaustively searching all possible messages which would be a second preimage search.

However, if attacker just tries to find any collision among the N birthdays the probability becomes

$$P_{Collision}(N) = 1 - \left(\frac{365}{365}\right)\left(\frac{364}{365}\right)\left(\frac{363}{365}\right)\cdots\left(\frac{365 - N + 1}{365}\right).$$

Here solving for N, it can be seen that $N = 23$ people are enough to have a match in the birthdays with probability greater than $1/2$.

One can generalize the "birthday case" to collision for hash function H. Let X be a set with $|X| = x$ and Q be a subset of X st $Q = \{q_0, q_1, \ldots, q_{t-1}\}$ with $H(q_i) = y_i$. The probability of two distinct elements $q_i \neq q_j$ of Q having the same hash value such that $H(q_i) = H(q_j)$ is

$$P_{Collision}(t) = 1 - 1 - e^{-\frac{t(t-1)}{2x}}.$$

This probability can be found using some approximations:

$$\begin{aligned}
P_{Collision}(t) &= 1 - \left(\frac{x}{x}\right)\left(\frac{x-1}{x}\right)\left(\frac{x-2}{x}\right)\cdots\left(\frac{x-t+1}{x}\right) \\
&= 1 - \prod_{i=1}^{t-1}\left(1 - \frac{i}{x}\right) \\
&= 1 - \prod_{i=1}^{t-1} e^{-\frac{i}{x}} \quad \text{since } 1 - \frac{i}{x} \approx e^{-\frac{i}{x}} \text{ for } i << x \\
&= 1 - e^{-\sum_{i=1}^{t-1}\frac{i}{x}} \\
&= 1 - e^{-\frac{1}{x}\sum_{i=1}^{t-1} i} \\
&= 1 - e^{-\frac{t(t-1)}{2x}}
\end{aligned}$$

The smallest value of t, which satisfies $P_{Collision}(t) = \frac{1}{2}$ can be found by solving the above approximation:

$$\begin{aligned}
P_{Collision}(t) = \frac{1}{2} &= 1 - e^{-\frac{t(t-1)}{2x}} \\
\frac{1}{2} &= e^{-\frac{t(t-1)}{2x}} \\
ln(\frac{1}{2}) &= ln(e^{-\frac{t(t-1)}{2x}}) \\
-ln(2) &= -\frac{t(t-1)}{2x} \\
2xln(2) &= t(t-1) \\
2xln(2) &\approx t^2 \\
\sqrt{2xln(2)} &\approx t \\
1,17\sqrt{x} &\approx t
\end{aligned}$$

Therefore, if attacker wants to find arbitrary collisions for an n bit hash function, she generates the hash value of approximately $2^{n/2}$ messages. There will be a collision with probability 1/2.

In the above collision search, the attacker searches for a collision on a single list of $2^{n/2}$ elements. Moreover, there are some attacks that uses two distinct lists of messages and searches for collisions of hash values of the elements from distinct lists. Then the attacker constructs two sets Q_1 and Q_2, and checks for any match between the hash values of elements of Q_1 and Q_2. The probability of such a collision is approximately equal to $1 - e^{\frac{|Q_1||Q_2|}{|X|}}$. If attacker chooses $|Q_1| = |Q_2| = \sqrt{|X|}$ then the probability will be approximately %63. This attack is called generalized birthday attack.

An example of generalized birthday attack is on signatures. Assume Eve wants to get the signature of Alice on an evil message M which Alice does not want to sign. She constructs a message M' that Alice is willing sign. Then Eve, by making minor changes in M and M', constructs $2^{n/2}$ variations of M and $2^{n/2}$ variations of M' and computes the hash values of all $2^{n/2+1}$ messages. With a probability greater than $1/2$ there will be a collision between the elements , say X and X', of the two sets. Then Eve asks Alice to sign X,which is a derivation of M that Alice is willing to sign. When Alice sends her sign on X, she will also be sending her sign on the evil message X'. So Eve, without any knowledge of Alice's private key or attacking to the signing algorithm, has obtained Alice's signature on evil message.

Note that the probabilities above are calculated with the assumption of the distribution of the birthdays, and respectively collisions, are uniform. If the distribution is not uniform then the probabilities will be higher. Therefore if the hash values are not uniformly distributed or there is a weakness in the structure of the hash function the probability of collision would be higher [65].

4.2 Correcting Block Attack

Correcting block attack is the most trivial second preimage tool which can also be used to find collisions.

The idea is, roughly, for a given message M_1 of t-blocks and corresponding hash value $H(M_1)$, producing a message M_2 of k-blocks which is shorter than M_1 and then trying to find message blocks $Y = m_c^1, \cdots, m_c^{t-k}$ such that $H(M_1) = H(M_2||Y)$.

In general, the attack is applied by correcting the last message block before padding and called correcting last block attack. In this case, attacker produces a message M_2 of length $t-1$ blocks. Let h_t and h'_{t-1} be, respectively, the final chaining values of messages M_1 and M_2 before processing padding blocks. Then the attacker searches for a message block m_c such that $f(h'_{t-1}, m_c) = h_t$ where f is the compression function of the hash algorithm. This way $H(M_2||m_c)$ will be equal to $H(M_1)$ and she will get a second preimage for M_1. The choice of message block to be corrected has no restriction and can be any message block but the first. If attacker chooses i^{th} message block to correct to find a second preimage, message blocks of M_2 after this message block should be equal to message blocks of M_1 after i^{th} message

blocks.

To find a collision, attacker randomly chooses two messages of the same length M_1 and M_2. Then, she searches for message blocks m_c and m'_csuch that $H(M_1||m_c) = H(M_2||m'_c)$ and gets a collision pair.

The complexity of finding such a correcting block is equal to exhaustive collision search for most of the modern hash functions and it is $O(2^n)$. However, some hash functions are easy to manipulate and this process can be less complicated. Especially hash functions based on modular arithmetic are vulnerable to correcting block attack.

An obvious countermeasure to be taken against correcting block attack is keeping the internal state size longer in order to increase the complexity of finding a correcting block. Moreover, using the message blocks two or more times will also thwart this attack but reduces the efficiency of the hash function.

4.3 Meet in the Middle Attack

The meet in the middle attack (MiMA) is a variant of birthday attack to find second preimages. In the birthday attack, attacker is comparing the hash values while in MiMA, she compares the internal chaining values. The advantage of MiMA over birthday attack is that MiMA enables attacker to construct a message with a desired hash value, ie., a second preimage. However, MiMA is applicable to hash functions with invertible compression functions where birthday attack can be applied to any hash function. In other words, to apply MiMA, attacker, knowing h_{i+1}, should be able to find a pair (h_i, m_i) st $f(h_i, m_i) = hi + 1$.

General idea of MiMA is going backwards from a hash value and going forwards from IV of the algorithm using some bogus message blocks, and then at a predefined meeting point trying to match them.

In [66], Nishimura and Sibuya defined three models for MiMA. In the first model, model A, attacker who knows M and $H(M)$, divides the bogus message M' into two parts. She chooses a chaining value as meeting point and starting from IV, generates k variations of the first part up to the meeting point. Then starting from the hash value $H(M)$, going backwards, generates $\overline{k}$ variations of the second part. In the figure each box is a chaining value and each arrow,

except those in the meeting point, denotes the message blocks.

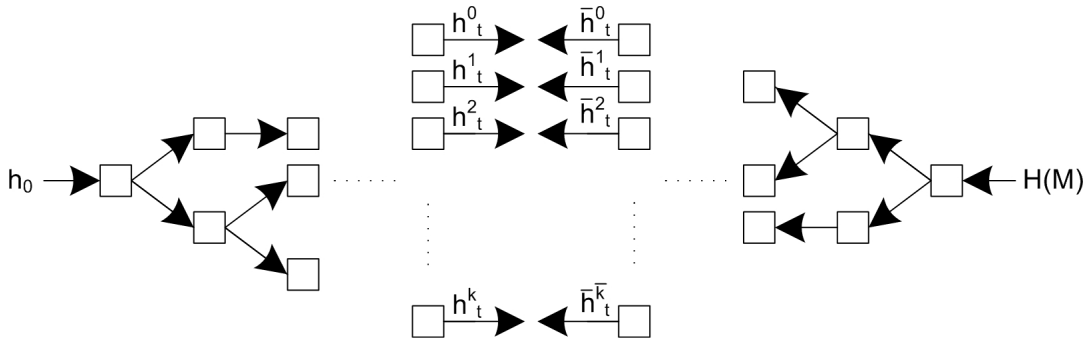

Figure 4.1: Model A

In the second model, model B, attacker, instead of changing the whole first part of the bogus message k times and the whole second part $\overline{k}$ times, she keeps fixed most of the bogus message. In this model, attacker goes forward from the IV up to the message block m_{t-1}, one stage before meeting point and goes backwards from hash value up to the message block m_t one stage after meeting point. Then generates k variations of m_{t-1} and $\overline{k}$ variations of m_t and checks for a match.

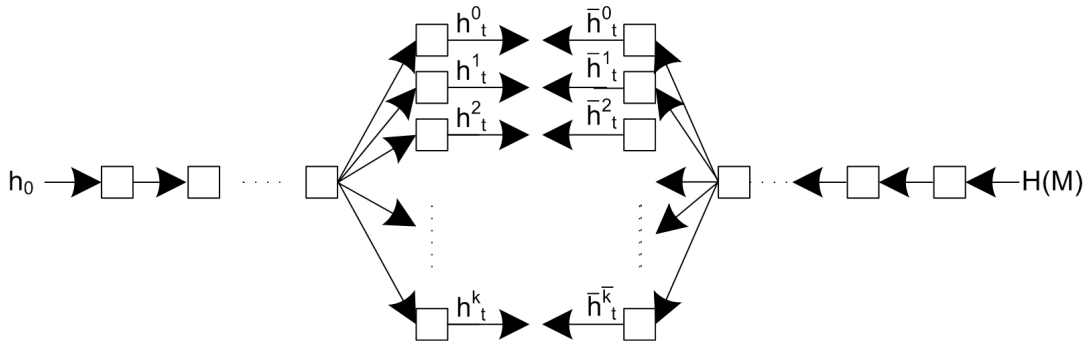

Figure 4.2: Model B

Model C is a synthesis of model A and model B. In model C, the first part of the bogus message is generated as in model A while the second part of the bogus message is generated as in model B. So the attacker goes forward from IV and generates k variations of the first part of the bogus message and goes backwards from hash value, $H(M)$, to the message block m_t, one step after the meeting point, and generates $\overline{k}$ variations of m_t.

The probabilities of all the above models are same as the generalized birthday attack. Taking

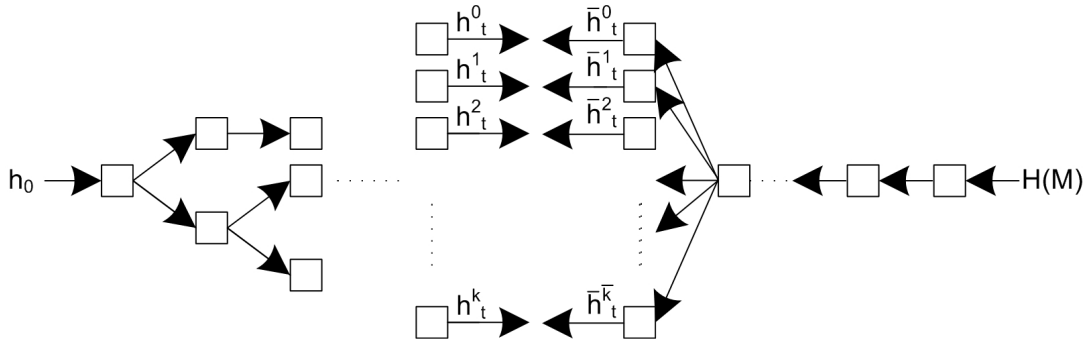

Figure 4.3: Model C

$k = \overline{k} = 2^{n/2}$ the probability of a match between h_t and $\overline{h_t}$ is about 1/2.

The above models and probabilities are applicable to hash functions which uses each message block at most once. Coppersmith [67] and Girault et al. [68] extended the attack to $p - fold$ hash functions which uses each message block p times.

The complexity of MiMA is roughly $2^{n/2+1}$. The exact complexities vary according to the message length and the model used in the attack. However, the cost is dominated by $2^{n/2+1}$.

To avoid MiMA, wide pipe or double pipe designs can be used. This makes finding collisions at meeting point cost more than $2^{n/2}$. Moreover designing hash functions which do not allow reversing compression functions will be a more efficient way to foil this attack.

4.4 Length Extension Attack

Length extension attack can be applied to the iterated hash functions by appending message blocks to the original or padded message. The aim of this attack is producing a hash value which is related to or contains a part of a message M without fully knowing it. The length extension attacks can be categorized in two as $Type\ A$ and $Type\ B$ attacks.

In $Type\ A$ attack, the hash function is assumed to use no MD-strengthening. For this type of attack consider two messages, $M_1 = m_0, m_1, \ldots, m_{t-1}$ and $M_2 = m_0, m_1, \ldots, m_{t-1}, m_t$ which are identical in the first t blocks. Then, when calculating the hash value of M_2, $H(M_1)$ appears as the chaining value which is input to the final iteration. Therefore $H(M_2) = f(H(M_1), m_t)$ where f is the compression function of H. This enables the attacker, without full knowledge

of M_1, to produce the hash value of a message M_2 which is related with M_1. One example of the attack is given in [69]. In an authentication protocol, which uses an iterated hash function without output transformation, two parties A and B share a secret X and the authentication protocol is as follows:

- A, generates some message M and sends $M, H(X||M)$ to B
- B, knowing X, compares $H(X||M)$ with the received data and authenticates A

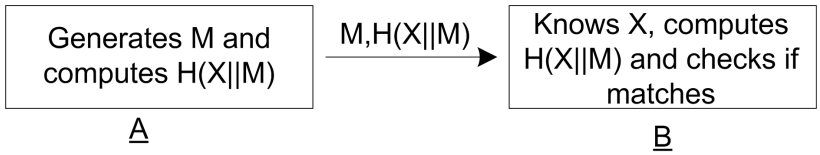

Figure 4.4: The Authentication Protocol

Here an attacker who knows M and $H(X||M)$ can impersonate A to B with extension attack. The attacker generates arbitrary message blocks $\overline{m_0}, \overline{m_1}, \ldots, \overline{m_{k-1}}$ and iterates $\overline{h_i} = f(\overline{h_{i-1}}, \overline{m_{i-1}})$ for $i = 1, 2, .., k$ with $\overline{h_0} = H(X||M)$. The result, $\overline{h_k}$, is the hash value of $M' = X||M||\overline{m_0}, \overline{m_1}, \ldots, \overline{m_{k-1}}$. Then the attacker sends M' and $\overline{h_k}$ to B. Party B, verifying the hash value, authenticates attacker as A.

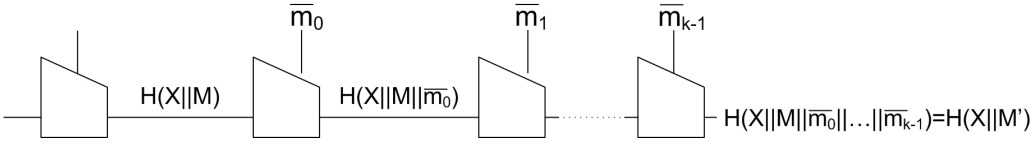

Figure 4.5: Expanding the Message

In *Type B* attack, hash function is assumed to use MD-strengthening. Considering the above protocol, this time attacker cannot extend the message by directly appending arbitrary message blocks and should know the length of $X||M$ or equivalently length of X. If she knows, L, the length of $X||M$, she knows the padding of $X||M$.

$$Pad(X||M) = X||M||1||00\ldots0||L$$

She considers $M' = M||1||00\ldots0||L$ so that $H(X||M)$ appears as a chaining value in $H(X||M')$. She applies padding to $X||M'$, $Pad(X||M') = X||M||1||00\ldots0||L||1||00\ldots0||L'$ and iterates

chaining values starting from $H(X||M)$ using the padding blocks $1||00\ldots0||L'$. Then sends M' and $H(X||M')$ to B. B, again, verifies the hash value and authenticates attacker as A.

There are several methods to foil the length extension attack. Using a proper output transformation is the most common precaution against this attack. If hash function applies an output transformation g to the final chaining values, then the hash value will be equal to $H(X||M) = g(h_t)$, where h_t is the final chaining value, and can not be used to extend the message. However, this transformation should not contain any fixed points, otherwise the attack can be applied to messages ending with the fixed points. Besides, in the final iteration the compression function may use different constants or counters as well.

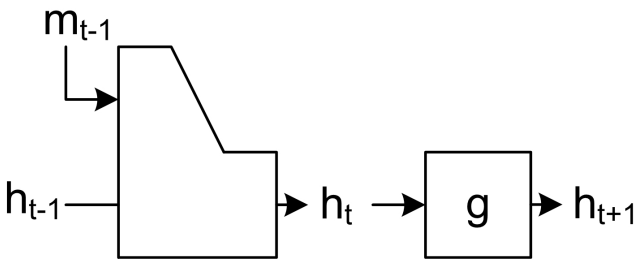

Figure 4.6: Output Transformation

Another method is, instead of appending the length padding after zero bits as a postfix, appending it in front of the message as a prefix [70]. This method, however, is not practical since for most of the applications the length of the message should be known before hashing.

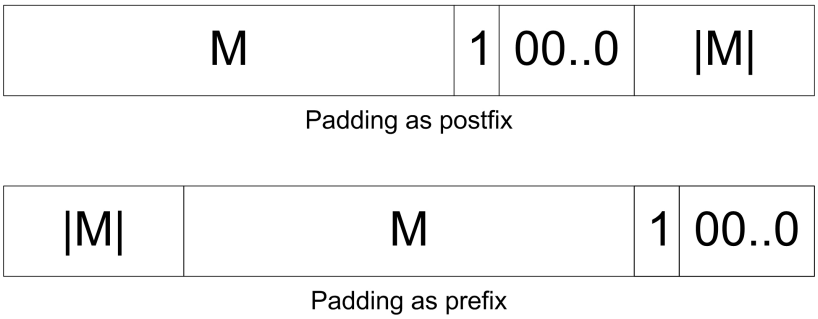

Figure 4.7: Prefix Padding and Postfix Padding

A third method is hashing the message twice or more. By hashing the hash of the data, attacker's control over the message will be minimized. Therefore, finding a proper message block to extend the message will be infeasible.

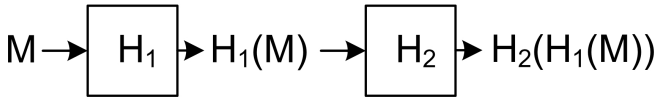

Figure 4.8: Double Hashing

4.5 Joux Attack

The presence of two distinct messages which have the same hash value is called a collision for a hash function H. Joux, in [54], extended the collision idea and introduced a new concept $multi-collision$ with $multi-collision\ attack$ which can be applied to iterated hash functions[1] . An *r-collision* for a hash function H is r distinct messages $M^1, M^2, \ldots, M^r$ with $H(M^1) = H(M^2) = \cdots = H(M^r)$. By Joux's multi-collision attack, the complexity of finding an $r-collision$ for an n-bit hash is reduced to $O(r2^{n/2})$ from $O(2^{n(r-1)/r})$. The attack also showed the security of cascaded hash functions are not as it thought to be.

To find an $r-collision$ first consider the messages of the same length. This results the padding of the messages to be equal. Therefore, if the chaining values before padding are equal, the hash values will also be equal. Another assumption is the presence of machine or algorithm C that can find "collisions" for the compression function f. This machine takes the chaining value h as an input and outputs two message blocks (m, m') such that $f(h, m) = f(h, m')$. C can use some weaknesses in f, birthday paradox or other collision attacks to find these pairs. For the time being assume C is using the birthday attack on f and finding a colliding pair for the compression function costs $2^{n/2}$ compression function computations. Besides, the below attack is for hash functions with message block length greater than or equal to the chaining value length. However, it can be generalized to any case easily which will be mentioned later in this section.

The attack is as follows: starting form the fixed IV of the hash function, attacker calls C

[1] The attack of Coppersmith to the Davies-Price variant of Rabin's hashing scheme can also be considered as multicolor attack.

iteratively t-times and gets the message block pairs (m_i, m_i').

$$h_0 = IV$$

for i=0,1,. . . ,t-1 do

$$(m_i, m_i') = C(h_i)$$

$$h_{i+1} = f(m_i, h_i)$$

This algorithm gives t pairs of message blocks, each is a collision for the compression function with the output chaining value of previous iteration. These messages can be used to construct 2^t distinct messages that have the same hash value.

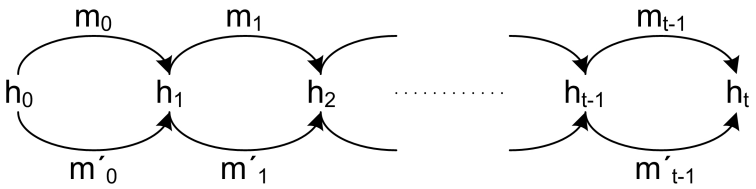

Figure 4.9: Multicollisions

To illustrate, assume $C(h_0) = (m_0, m_0')$, $f(h_0, m_0) = h_1$, and $C(h_1) = (m_1, m_1')$, $f(h_1, m_1) = h_2$. Then the messages $m_0||m_1$, $m_0||m_1'$, $m_0'||m_1$ and $m_0'||m_1'$ will all have the same final chaining value h_2. After padding, since the lengths of the messages are equal, the hash values of these messages will also be equal. Moreover, all the chaining values appeared in the hashing processes of all the four messages will be the same.

If the length of the chaining value is greater than the message block length, attacker, instead of finding collisions for each iteration of hash function, can try to find collisions for a number of consecutive iterations. Let k be the smallest number such that $k|m| \geq |h|$. Then she tries to find colliding pairs of messages of k blocks

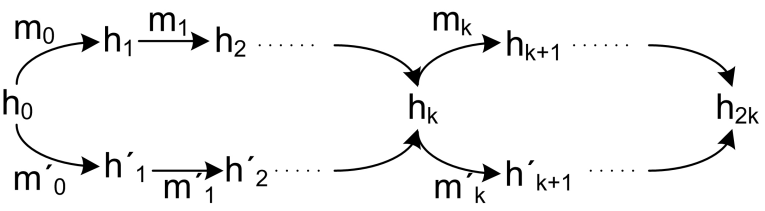

Figure 4.10: Multicollisions for $m < h$

Let the message strings be $S_0 = m_0, m_1, \ldots, m_{k-1}$, $S'_0 = m'_0, m'_1, \ldots, m'_{k-1}$, $S_1 = m_k, m_{k+1}, \ldots, m_{2k-1}$ and $S'_1 = m'_k, m'_{k+1}, \ldots, m'_{2k-1}$. Then the four messages of length $2k$, $S_0||S_1$, $S_0||S'_1$, $S'_0||S_1$ and $S'_0||S'_1$, will have the same hash values.

The complexity of the attack is t times the complexity of calling C, which is, $O(t2^{n/2})$.

This attack can be applied to the cascaded hash functions also. A cascaded hash function is the concatenation of two or more hash functions. Let H_1 and H_2 be two hash functions of l_1 and l_2 bits output respectively. Then $H(M) = H_1(M)||H_2(M)$ is a cascaded hash function with $l_1 + l_2$ bits output. The attack described here is on the concatenation of two hash functions but can be generalized directly to concatenation of any number of hash functions and shows that finding a collision for a cascaded hash function is $O(l_2 2^{l_1/2} + 2^{l_2/2})$ instead of $O(2^{(l_1+l_2)/2})$.

Let H, H_1, and H_2 be as above with $l_2 \geq l_1$. Attacker takes $t = \lceil \frac{l_2}{2} \rceil$ and constructs 2^t colliding messages for H_1 using the above multi-collision algorithm. Since $t \geq \frac{l_2}{2}$, by birthday paradox, there exist a collision for H_2 in the set of 2^t colliding messages of H_1 with probability greater than 1/2. So, by generating 2^t multi-collisions for H_1 attacker will get a collision for H with probability $1/2$.

The complexity of this attack is $O(l_2 2^{l_1/2} + 2^{l_2/2})$. The $O(l_2 2^{l_1/2})$ is the complexity of the multi-collision on H_1. The rest $O(2^{l_2/2})$ comes from the birthday paradox. The complexity of hashing 2^t messages of t blocks with H_2 is reduced from $t2^t$ to 2^t by using the tree structure of the messages.

Here, H_1 and H_2 are assumed to work on the same block sizes. An important point is that, since the attacker applies birthday attack on H_2, it need not to be iterative, on the contrary, can be any type of hash function.

Same idea can be used to find preimages and second preimages for the cascaded hash functions. For preimage attack attacker, again, uses the multi-collision algorithm with $t = l_2$ and constructs 2^t colliding messages M_i for H_1. Then, considering the padding of the messages, searches for an additional message block that maps the final chaining values of H_1 to a specific target hash value N. Now the attacker has 2^t distinct messages with the specific hash value N on H_1. Since $t \geq \frac{l_2}{2}$ with constant probability the hash value, $H_2(M_i)$, of one of these messages also matches with the specific hash value on H_2 and a preimage for a random hash value can bu found. If the attacker specifies the hash value, than she can find a second

preimage.

Another way to find a preimage ,or second preimage, is to construct 2^t colliding messages and search for a $(t+1)^{th}$ block which gives the desired hash value [57]. Again one should take the padding into account when searching for such a message block.

Joux's multi-collision attack does not apply directly to the hash functions that use message blocks more than once. However, Nandi and Stinson applied the attack to the hash functions which uses message blocks at most twice [71]. Then Hoch and Shamir extended the idea to the hash functions that uses message blocks multiple times and in an arbitrary expanded order [72].

To foil Joux multi-collision attack double or wide pipe designs with the size of the internal chaining values $> 2n$ can be used. This way cost of collisions will exceed the exhaustive search.

4.6 Fixed Point Attack

A fixed point for the hash function H is a pair (h, m) such that $f(h, m) = h$, ie, the chaining value h remains the same after iteration with message block m. Fixed points are not just the chaining values or the message blocks: h which is a fixed point for the message block m will not form a fixed point pair with another message block, or vice versa.

Fixed points are used to attack iterated hash functions. Let the pair $(h_i, \overline{m_i})$ be a fixed point pair for the compression function f of H. Consider the message $M = m_0, m_1, \ldots, m_i, \ldots, m_{t-1}$ with chaining values $IV = h_0, h_1, \ldots, h_i, \ldots, h_t$ appear when hashing M.

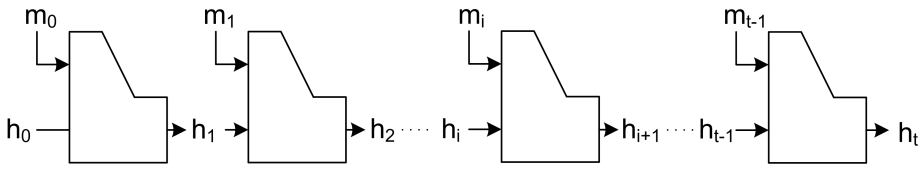

Figure 4.11: Hashing Process of M

Attacker can construct another message $M' = m_0, m_1, \ldots, m_{i-1}, \overline{m_i}, m_i, \ldots, m_{t-1}$. which produces the chaining values $IV = h_0, h_1, \ldots, h_{i-1}h_i, h_i, h_{i+1}, \ldots, h_t$.

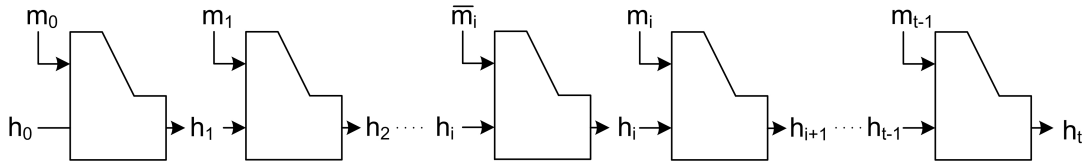

Figure 4.12: Hashing Process of M'

Since $(h_i, \overline{m_i})$ forms a fixed point pair, insertion of $\overline{m_i}$ to the iterations where h_i is the input, will not change the final chaining value. So the attacker can insert as many $\overline{m_i}$ message blocks as she needed without changing the hash value to find a second preimage. However, the problem with this attack is the length padding. As she inserts message blocks to produce a second preimage, the message M' will be longer than the original message M. Therefore the length paddings of two messages will be different and the hash values will not be the same.

In [53], Dean proposed three methods to by-pass the length padding problem:

1. The first idea uses a flow in the length padding of particular hash functions. In some hash functions ,like $MD-4$, $MD-5$ and $SHA-1$, the length of the message modulo a number, say L, is padded. So if the attacker inserts message block $\overline{m}_i$ $\frac{L}{block\ length}$ times, which adds L bits to the message, the length of the new message and original message will be equal modulo L. Therefore the length padding will be the same as the original message. In cases of mentioned hash functions above, length of the message modulo 2^{64} is padded therefore the message M', produced by repeating the message block $\overline{m_i}$, which is 512 bits, 2^{55} times, will have the same hash value with M.

2. Assume that after two distinct iterations, say k^{th} and l^{th}, the resulting chaining values are the same: $h_k = h_l$ with $k < l$. and there is a chaining value h_i ,which is a fixed point for the message $\overline{m}_i$, that does not lie between h_k and h_l.

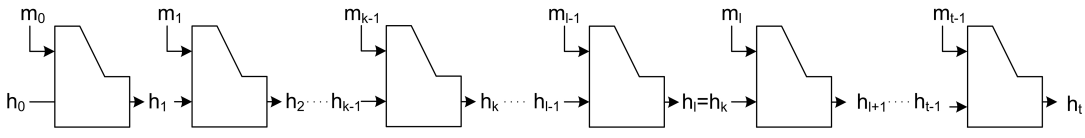

Figure 4.13: Coincidence in the Chaining Values h_k and h_l

Then since $h_k = h_l$, attacker deletes the message blocks $m_k, m_{k+1}, m_{k+2}, \ldots, m_{l-1}$ from

the message M. The shrunk message $\overline{M}$ has the same final chaining value, up to the padding, as the original message M but it is $(l-k)$ blocks shorter than it. For the length issue she adds message block $\overline{m}_i$, after the iteration that outputs h_i, $(l-k)$ times. Again this operation does not change the final chaining value and $\overline{M}$ becomes of the same length with M. Therefore, the hash values of these messages will be the same.

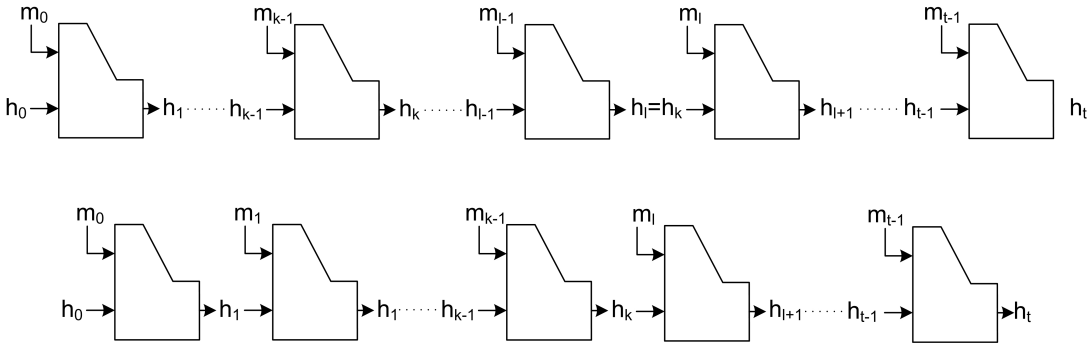

Figure 4.14: Shrinking the Message

3. If there is no coincidence in the chaining values, attacker tries to modify some message block m_{k-1} such that $h_k = f(h_{k-1}, m_{k-1})$ is equal to some other chaining value. Then the same procedure as in second method can be applied to get a second preimage.

Finding a fixed point for some specific constructions is easier than the generic methods. For example Davies-Meyer construction in *Figure 4.15*:

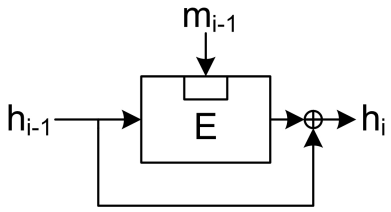

Figure 4.15: Davies-Meyer Construction

Here if $E_{m_i}(h_i) = 0$ than (h_i, m_i) will be a fixed point pair. So, one decryption operation is enough to find a fixed point pair (h_i, m_i).

$$E_{m_i}(h_i) = 0$$
$$\Rightarrow \quad E^{-1}_{m_i}(0) = h_i$$

Also Snefru [73] has a similar weakness. The compression function of Snefru is an encryption process $f(h, m) = lsb_n(E(h||m)) + h$, where $lsb_n(x)$ denotes the least significant n bits of x. If $lsb_n(E(h||m)) = 00$ then one can construct a fixed point pair easily. To construct a pair, attacker chooses any value T with $lsb_n(T) = 0$ and lets $T = E(h||m)$. Then decrypting T, she gets $E^{-1}(T) = Y = h||m$. The least significant b bits of Y is m, where b is the message block length and the most significant n bits of Y is the chaining value h. These values of h and m satisfies $f(h, m) = h$.

$$m = lsb_b(Y) \qquad h = msb_n(Y)$$

Fixed points, moreover, are used in various attacks to mount the attack or make it more efficient. Fixed points are used in the attacks on Gost Hash, RC4 Hash [15] etc, and in herding attack, Kelsey and Schneider's second preimage attack.

The cost of fixed point attack is approximately equal to the cost of finding fixed point pairs which changes according to the compression function. Moreover, if padding is applied then this cost increases.

The basic defense against fixed point attack is using counters. This way attacker could use a fixed point triple $(h_i, \overline{m_i}, c)$ only once. However, even she could inject message block $\overline{m_i}$ properly, the counters of the following iterations will change and the all the following chaining values, including the hash value, will change. Moreover, salt, which is used with each message block according to the iteration number, can also prevent the attacker from using fixed points. Another method is using message blocks more than once. The message block $\overline{m_i}$ which produces a fixed point pair with h_i, will not produce another pair in other iterations that it will be used. This can also be achieved by following a double pipe design.

4.7 Long Message Attack

The long message attack is a second preimage attack using long messages for iterative hash functions. Attack is applicable to messages of any length but it is more efficient when the original message is very long.

Assume the attacker is trying to find a second preimage for the message $M = m_0 m_1 m_2 \dots m_{l-1}$ which produces the chaining values $h_0 = IV, h_1, \dots, hl - 1, h_l = H(M)$ when hashing.

Attacker produces a prefix $\overline{M} = \overline{m_0 m_1} \ldots \overline{m_{k-1}}$ with final chaining value $f(\overline{h_{k-1}}, \overline{m_{k-1}}) = \overline{h_k}$. If $\overline{h_k}$ is equal to one of the chaining values $h_1, h_2, \ldots, h_{t-1}$, the attacker can bind the message blocks of the original message after the matching chaining value to $\overline{M}$. Let $\overline{h_k} = h_a$, then $\overline{M}$ will be $\overline{M} = \overline{m_0 m_1} \ldots \overline{m_{k-1}} m_a m_{a+1} \ldots m_{l-1}$ and will have the same hash value with M. If $\overline{h_k}$ is not equal to any of the chaining values, then attacker tries to find a linking message m_{link} to produce another chaining value $\overline{h_{k+1}}$ which is equal to one of the chaining values. Since there are $l - 1$ chaining values appearing in hashing process of M, assumed to be all distinct, $\overline{h_k}$, or $\overline{h_{k+1}}$, can be equal to one of these chaining values with probability $\frac{l-1}{2^n}$. Letting $l = 2^b$, the cost of a match will be 2^{n-b} hash function computations.

The final chaining values of M and $\overline{M}$ are the same. However, since length padding is processed at all modern hash functions , hash values of these messages can be different depending of their lengths. Let the matching chaining values of these messages be $h_a = \overline{h}_k$. Then there are three situations:

- If $k < a$, then $\overline{M}$ is shorter then M. Then the attacker can use the methods used in the fixed point attack, in Section 4.6, proposed by Dean [53] to extend the message $\overline{M}$.
- If $k = a$, then the lengths of the messages will be equal. Therefore the hash values $H(M)$ and $H(\overline{M})$ will be equal.
- If $k > a$, then $\overline{M}$ is longer than the original message M. This time attacker should use the tricks in the fixed point attack to shrink the message to the same length with M.

Moreover, Kelsey and Schneier developed an attack, that will be explained in the next section, which can be applied using the long message attack.

4.8 Kelsey & Schneier's Long Message Attack

In 2005, Kelsey and Schneier proposed a new second preimage attack which is essentially similar to Joux multi-collision attack and integrated with the long message attack [55]. The attack can be combined with Dean's fixed point idea for hash functions that are easy to find fixed points.

Attack is basically, as Joux's attack, depends on finding successive collisions. The main

difference of this attack from the Joux multi-collision is, instead of finding colliding pairs of one block messages, attacker searches for pairs of collisions of different lengths. This way she can construct a message having a fixed hash value but that can vary in length. Consider this message as a single elastic message. This elastic message is called an *expandable message* and an expandable message that can take any length between a and b blocks is called an $(a, b) - expandable\ message$.

Finding collisions for different length messages can be achieved in two ways. First method is using the fixed points. In this method, attacker generates $2^{n/2}$ fixed point pairs (h_i^f, m_i^f) and $2^{n/2}$ chaining values h_i^1 that can be reached from IV using message block m_i^0. With a high probability there will be a match between h^f's and h^1's. Let $h_i^1 = h_j^f$, then the messages m_i^0 and $m_i^0||m_j^f$ will produce the same final chaining values. Attacker can also append as many m_j^f blocks to the second message as she wants. This method is only applicable to hash functions which are easy to find fixed points. Therefore will not be used as the initial method in this attack.

The second method, despite having more time complexity, is more practical since it can be applied to any iterated hash function. In this method, to find two colliding messages one is of length one block and other is of length t blocks, given an initial value h_{in}, not necessarily $h_{in} = h_0$, attacker first produces a dummy message $\overline{M}$ of length $t-1$ blocks, using a dummy message block d, as $\overline{M} = dd \dots d$. Then generates $2^{n/2}$ chaining values that can be reached from h_{in} by a single message block $h_i^1 = f(h_{in}, m_i^0)$ and $2^{n/2}$ chaining blocks that can be reached from the final chaining value of the dummy message $\overline{M}$ using a single message block $h_i^t = f(h_{t-1}, m_i^{t-1})$. With a non-negligible probability there will be a match between the chaining values h^1 and h_i^t. If $h_i^1 = h_j^t$, then the two messages m_i^0 and $\overline{M}||m_j^t$ will have the same final chaining values. The complexity of finding such a pair is $t - 1$ compression function calls for the dummy message and two $2^{n/2}$ compression function calls to find a match between chaining values. The total complexity of this algorithm is $t - 1 + 2^{n/2+1}$ compression function calls.

Using the second method attacker can generate an expandable message. To build a $(k, k + 2^k - 1) - expandable\ message$ attacker first produces a pair of messages of lengths 1 and 2 as above. Then she produces another pair with messages of lengths 1 and 3. This way she produces all pairs each having messages of lengths 1 and $2^{i-1} + 1$ for $i = 1, 2, \dots k$. The key point is that the attacker uses the final chaining value of the previous pair as the initial value

of the pair to be generated. Therefore the message pairs can be concatenated to produce an expandable message.

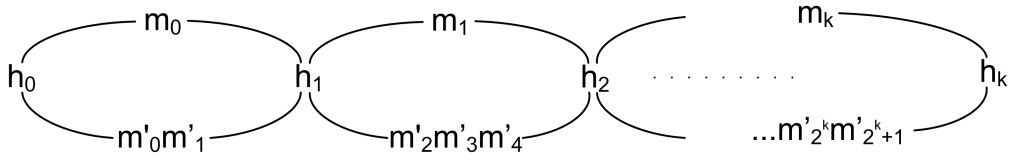

Figure 4.16: A $(k, k + 2^k - 1)$-Expandable Message

The produced expandable message can take any length between k and $k + 2^k - 1$ blocks. To produce a message of desired length L, $k \leq L \leq k + 2^k - 1$, attacker uses the expandable message and the bitwise notation of $L - k$. Attacker reverses the k-bit notation of $L - k$ and stores it. For example, take $k = 10$ and $L = 16$ then $L - k = 6$. The 10- bit notation of 6 is 0000000110 and when reversed it becomes 0110000000. Attacker, starting with an empty message, appends one of the messages of the corresponding collision pair according to the reversed bitwise notation. If the i^{th} digit of the reversed bitwise notation of $L - k$ is 0, then she appends the single block m_i of the pair. Otherwise she appends the $2^i + 1$ block message of the pair. The resulting message will be of length L and will have the fixed hash value of the expandable message.

To illustrate, assume the attacker constructed a $(2, 5) - \textit{expandable message } \overline{M}$ as in Figure 4.17.

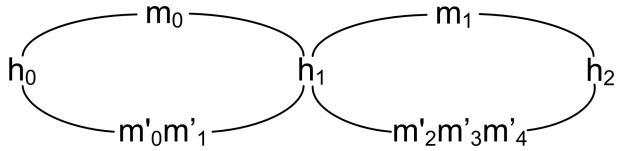

Figure 4.17: $(2, 5)-$Expandable Message

Let $L = 4$. Then $L - k = 2$ which have 2-bit notation 10 which becomes $R = 01$ when reversed. Since the first element of R is 0, attacker chooses the one block message m_0 of the first iteration. The second element of R is 1, therefore, this time she chooses 3 block message of the second pair $m'_2 m'_3 m'_4$. Concatenating the two mesages she gets $M' = m_0 m'_2 m'_3 m'_4$ which is of length 4 and has hash value h_2.

The constructed message has a fixed hash value for various lengths but, itself, cannot be regarded as a bunch of second preimages since hash functions use length padding. Therefore, additional tools should be used with expandable messages. In [55], Kelsey and Schneier applied the long message attack with expandable message to bypass the length padding in the hash functions.

The attacker, aiming to find a second preimage for a long message M, first generates a $(k, k + 2^k - 1) - expandable\ message$ which has a final chaining value, say, $\overline{h_k}$. Then, if $\overline{h_k}$ is not equal to any of the chaining values that appears when hashing M, appends a linking block m_{link} such that $f(\overline{h_k}, m_{link})$ is equal to one of those chaining values. Otherwise, she continues without appending any message blocks. Let $M = m_0 m_1 \dots m_l$ and the chaining values appearing when hashing M be $h_0, h_1, \dots, h_{l+1}$, where l is close to maximum possible value, and let $\overline{M}$ be the expandable message with final chaining value $\overline{h_k}$. If $\overline{h_k} = h_j$ for some j with $k \leq j \leq k + 2^k - 1$, then attacker expands $\overline{M}$ to j blocks and appends the message blocks $m_j, m_{j+1}, \dots, m_l$ to $\overline{M}$. So $M' = \overline{M} || m_j m_{j+1} \dots m_l$ will have the same final chaining value and length therefore the same hash value with M. If $\overline{h_k}$ is not equal to any of the chaining values then attacker searches for a linking message block m_{link} such that $f(\overline{h_k}, m_{link}) = h_{k+1} = h_j$ where $k + 1 \leq j \leq k + 2^k$. Then she extends M to $j - 1$ blocks, appends m_{link} and the message blocks $m_j, m_{j+1} \dots, m_l$ to $\overline{M}$ and gets $M' = \overline{M} || m_{link} || m_j m_{j+1} \dots m_l$ which is a second preimage for M.

The total cost of the attack is approximately the cost of constructing expandable message plus the cost of finding a linking message m_{link}. A $(k, k + 2^k - 1) - expandable\ message$ can be constructed with $3 \times 2^{n/2+1}$ compression function calls using fixed points and $k \times 2^{n/2+1}$ compression function calls using generic method. Finding a linking message, if necessary, is 2^{n-k+1}. So the total complexities are $3 \times 2^{n/2+1} + 2^{n-k+1}$ for fixed point approach and $k \times 2^{n/2+1} + 2^{n-k+1}$ for generic approach.

This attack is more efficient than Joux multicollision attack for extremely long messages but for shorter messages Joux attack is much more applicable.

Using counter is a method to thwart this attack. When attacker produces expandable message, she cannot consider counters since collisions are of distinct lengths. Also increasing the internal state with double or wide pipe designs will make the complexity of finding collisions larger.

4.9 Herding Attack

Kelsey and Kohno developed an attack on Merkle-Damgård hash functions in [74], where the attacker first reveals the hash value and then produces a message which has the revealed hash value. Since the attacker produces the message after finding its hash value, she can prove prior knowledge of any information. Therefore, the attack is also called Nostradamus attack.

An example to clarify the attack can be as follows: Attacker claims that she knows the result of a match which will be played soon and provides a hash value which she declares as the hash value of the result of the match. After the match, she publishes a message containing the result of the match which has the previously declared hash value.

This attack has two phases: first the attacker constructs a diamond structure which contains colliding chaining values. These chaining values reduce to a single chaining value h_t and attacker releases h_t.

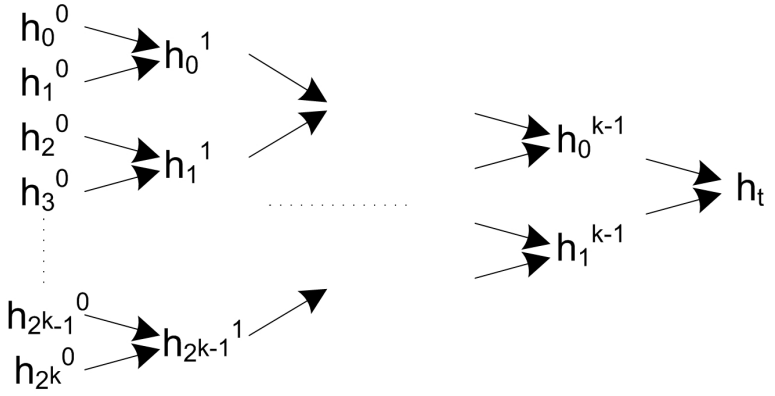

Figure 4.18: Diamond Structure

In Figure 4.18 a diamond structure is depicted where each arrow denotes a message block which are not necessarily distinct.

After learning the result of the match, which will be called the prefix P, she finds a linking message m_{link} that links the final chaining value of the prefix to the first column of the diamond structure.

The diamond structure is consisting of colliding messages. To construct a diamond structure of width k one should find 2^k multi-collisions. The idea of diamond structure is similar to

Joux multi-collision however there are two main differences:

1. The diamond structure allows 2^k choices for the first message block and the rest of the messages are determined by the first block while Joux attack provides two message block choices for each iteration.

2. Diamond structure contains $2^{k+1} - 2$ chaining values. Joux 2^k-multi-collision has k chaining values.

The first difference makes finding a linking message more efficient and the second enables the attacker to apply herding attack to short suffixes.

To produce the structure, attacker searches for collisions dynamically. In other words, the positions of the chaining values are not fixed, attacker iterates chaining values with arbitrary message blocks and orders the chaining values as she finds collisions. This way, in the first step, to reduce the 2^k chaining values to 2^{k-1} chaining values, attacker needs $\sqrt{2^{n-(k+1)}}$ message blocks. Therefore, for a full diamond structure she needs $\sqrt{2^{n+k+4}}$ message blocks. Also, attacker does not have to construct a diamond structure for each attack, on the contrary a diamond structure can be used for various herding attacks but the revealed has value will always be same or related.

To find the collisions more efficient, collision finding algorithms or weaknesses in the algorithm can be used. However, instead of collisions with a fixed IV, attacker needs collisions each starting with a different IV.

After constructing the diamond structure, attacker publishes the target chaining value which all 2^k chaining values reduce to. Then, when the prefix P is determined, she searches for a linking message m_{link} which links the final chaining value h_p of P to one of 2^k chaining values in the first column of the diamond structure. This costs 2^{n-k} compression function calls. If, instead of diamond structure, Joux multi-collision attack were used, finding a linking message would cost 2^{n-1} compression function calls. Assume the attacker finds a message m_{link} such that $f(h_p, m_{link}) = h_i^0$ for some $i = 0, 1, \ldots 2^k - 1$. Tracing from h_i^0 to h_t, one can generate a sequence S st the message $P||m_{link}||S$ has the published hash value.

If the hash function uses MD strengthening, she can guess the length of the prefix P. For the match example above, the names of the teams are known and the scores will most probably

be one digit numbers so the length of P can be guessed. The total length of the message will be $l = |P| + 1 + k$ where "1" comes from the linking message and k is the length of S. Iterating the target hash value h_t with m_{pad}, padding for a message length l, gives the new target hash value. Another approach is keeping the length l of the message much longer than enough and add some meaningful text. This time instead of "'*The result of the match is ..*"', the prefix can be extended like "'*Me, the Nostradamus of modern times, foresaw that ..*"'. Moreover the attacker does not have to clearly state the result of the match, she can make a general comment about the result as "'*Team A wins*"' or "'*Team A sweeps B*"'. This type of attack costs $2^{n-k} + 2^{n/2+k/2+2}$ which is the cost of finding a linking message plus constructing the diamond structure.

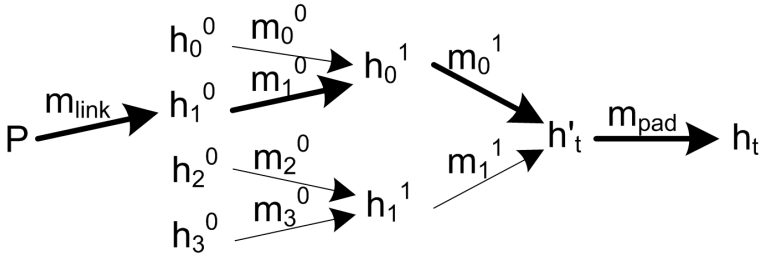

Figure 4.19: Sample Attack (m_{pad} is the padding block for guessed length)

To illustrate the attack, in the Figure 4.19, attacker constructing a diamond structure with width 4, finds a linking message that links h_p to h_1^0. Then traverses the structure with string $S = m_1^0||m_0^1$ and gets the message $M' = P||m_{link}||S$. When hashing the message the final chaining value of M' will be h'_t. Adding the padding block, m_{pad}, one reaches the target hash value h_t.

Moreover, using a $(log(k), k+log(k)-1)$-expandable message, attacker can search for a linking message that links h_p to not only the 2^k chaining values in the fist column but any of the $2^{k+1}-2$ chaining values in the diamond structure and expand the message so that S is always of fixed length.

If attacker uses expandable message in the diamond structure, she will be more flexible about the length padding. Assume attacker constructs an expandable message which she can extend k blocks. Then, she can guess the length of the prefix with an error up to k blocks with the same hash value. This way, instead of finding a linking message that links the final chaining

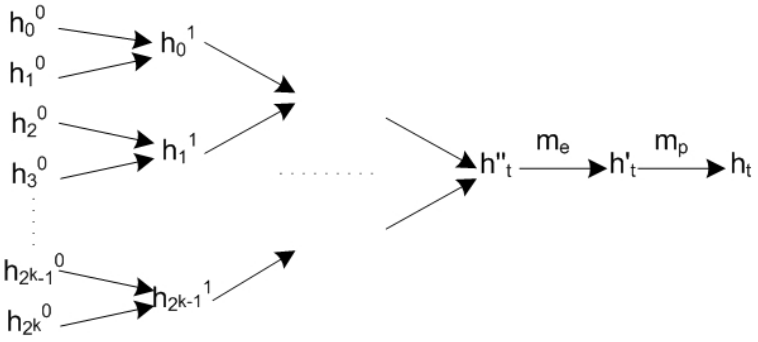

Figure 4.20: Diamond Structure with an Expandable Message

value of the prefix, h_p, to the first column of the diamond structure with 2^{n-k} complexity, attacker searches for a linking message which links h_p to any of the $2^{k+1} - 2$ chaining values in the structure with a complexity 2^{n-k-1}. The attacker first constructs a diamond structure in which the chaining values reduce to a single chaining value h_t''. Then she constructs a $(log(k), k + log(k) - 1)$-expandable message m_e which brings h_t'' to h_t'. She applies padding iterations to h_t' and reveals the final chaining value h_t.

When the prefix is determined, if necessary, she finds a linking message m_{link} that links h_p to one of the chaining values in the diamond structure. Let $f(h_p, m_{link}) = h_i^j$. Then she traverses the structure from h_i^j to h_t'' and finds the string S. Then according to the prefix length she expands m_e as needed say q blocks. The message that has the desired hash value h_t is

$$M' = P||m_{link}||S||m_e^q.$$

The cost of the herding attack that uses expandable message is the cost of producing an expandable message plus the cost of constructing diamond structure and the cost of finding a linking message which adds up to $k \cdot 2^{n/2+1} + 2^{n/2+k/2+2} + 2^{n-k-1}$.

The optimal herding attack can be mounted by using an expandable message and choosing k such that, the work done for constructing the diamond structure and finding the linking message block is equal.

In [75] Dunkelman and Preneel extended the herding attack to concatenated hashing schemes. The attack on concatenated schemes uses multiple diamond structures. For a hash algorithm which is constructed by concatenating k hash functions the complexity of herding attack with

a diamond structure of width 2^t will be approximately

$$2^{n_1-t} + \sum_{i=2}^{k}(n_i - t)\prod_{j=2}^{k-1} 2^{n_k-t}$$

where n_i is the chaining size of i^{th} hash function.

4.10 Slide Attack

Slide attack is a cryptanalysis tool for block ciphers with weak key schedule, which was introduced by Wagner and Biryakov in [76]. However, in [77] Gorski, Lucks and Peyrin showed that the attack is also a threat for stream based and sponge hash functions. By applying slide attack to hash functions one can distinguish the algorithm from a random oracle and mount an extension attack similar to the length extension attack for MD hash functions in Section 4.4.

Slide attack is applicable to block ciphers which have a weak or periodic key schedule. These block ciphers can be divided into equivalent permutations according to the round keys. For example, let the key size of a 16 round block cipher E is 128 bits and i^{th} round key k_i is produced by rotating the main key left by $i * 32$. Then the round keys will repeat after 4 rounds and will be equal at each four rounds. Let F be the permutation equal to the first four rounds of E. Then, one can consider E as $E(P) = F \circ F \circ F \circ F(P)$. If the corresponding ciphertexts C_1 and C_2 of plaintexts P_1 and P_2, with $P_2 = F(P_1)$, satisfies $C_2 = F(C_1)$ then (P_1, P_2) is called a slid pair. Using the slid pairs attacker can retrieve information about the key and distinguish the cipher from a random oracle.

In hash function case, one can consider the blank rounds at the end of stream and sponge based hash functions as identical permutations which does not take any input apart from the chaining value and applies a fixed permutation on it. So, parallel to these consideration, the definition of a slid pair will change. Two messages M_1 and M_2 are called a slid pair if $S_2 = B(S_1)$ where $B(\cdot)$ denotes a blank round and S_1 and S_2 are the final states appear before the output transformation, when hashing these messages. There is no constraints on the messages as in block cipher case.

If the state after processing the first blank round of M_1 is equal to the state before the blank rounds of M_2, (M_1, M_2) will be a slid pair.

Assume that the attacker is trying to attack the protocol as in Figure 4.4 of Section 4.4. Stream and sponge hash functions apply an output transformation to the state (or states) after blank rounds. Therefore, in such a protocol, attacker who knows $H(K||M)$ and M cannot recover the final state and append new message blocks directly. For example, in Grindahl [78], the final state is truncated according to the hash size. So, in order to extend the message attacker needs to find the full final state. Slide attack enables the attacker to find the final state with lower complexity.

In this section the attack is applied on Grindahl[2] hash function. The specifications of Grindahl-512 can be summarized as follows

- Message M is padded and divided into 64-bit blocks $m_0 m_1 \ldots$.
- An initial state is constructed as a 8x13 matrix whose entries are all zero.
- i^{th} message block is replaced with the first column of the i^{th} state. Then some permutations are applied to the overwritten state.
- 8 blank rounds, which are consisting of permutations only, are applied to the state after all the message blocks are input. In these blank rounds state is neither inserted any message nor truncated.
- The first 8 columns of the state after blank rounds are taken as the output. In other words the final 5 columns are truncated.

Now, the attacker can try to mount an extension attack to the protocol in Figure 4.4 by knowing M and $H(K||M)$ using the slide technique. Let the first message be $M_1 = m_0 m_1 \ldots m_{t-1}$. Let

$$Pad(K||M_1) = K||m_0 m_1 \ldots m_{t-2}||m_{t-1} + 10...0||P_L$$

where P_L is the length padding. Attacker produces a message M_2 such that $Pad(K||M_1)$ and $Pad(K||M_2)$ differs only in one additional block at the end.

Assume the attacker has an access to a query machine which outputs the hash value $H(K||M)$ when input M. She sends M_1 and M_2 to the machine and gets $H(K||M_1)$ and $H(K||M_2)$ which are the truncated values of the final states after blank rounds. By using these hash values she

[2] Grindahl hash function is not considered to be a sponge hash function by the authors who proposed sponge construction[56]. However, in cryptographic community most interested authors threat this hash function as a sponge.

can recover the 8 columns of the final states. However, there are still 5 columns which are unknown to the attacker.

First, she tries to find slid pairs. M_1 and M_2 differs only on one block, therefore, there are 2^{64} possible messages M_2 only one of which is the slid-mate of M_1. To find a slid pair, attacker constructs S_F^1, the final state of $K||M_1$, and S_F^2, the final state of $K||M_2$, using the hash values and fixing the unknown bytes. Then applies an inverse blank round to S_F^2 and compares the known-bytes with S_F^1. The 34 bytes will be known in both states. Using these bytes attacker can eliminate $2^{34*8} = 2^{272}$ wrong pairs. Since there are possible 2^{64} pairs, the slid pair (M_1, M_2) can be found with a very high probability.

In the state, produced by an inverse blank round from the hash value of $K||M_2$, there are also 20 known-bytes which coincide with unknown bytes in S_F^1. Therefore, these 20 bytes can be replaced with unknown bytes in S_F^1 where attacker left with 10 unknown bytes. To recover these 10 bytes, she goes backwards through all 8 blank rounds and the round before the blank rounds, up to replacement of the final message block, by trying all possible values of 10 bytes and checks the first column of the states. If the column is equal to the final block of $Pad(K||M_1)$, which is the length of $K||M_1$ that is assumed to known, then the choices of the 10 bytes and, therefore S_F^1, is true. Now, attacker knows the state before blank rounds and can extend the message $K||M_1$. For example, let $M_3 = Pad(K||M_1)$. Attacker who wants to hash M_3, knows the state before the padding of M_3 and can continue to process hashing. Then she sends $M = M_1||10..0||P_L||10..0||P'_L$ and $H(M_3)$ which authenticates her to Alice.

Moreover, instead of extending a specific message, she can try to go backwards until the last state where the key blocks are input. This procedure is not obvious because of the replacement operation. As she finds the final state, attacker can try to guess some bytes which led to corresponding message blocks in the first column as above. This way, she can find the value of $H(K||M)$ for any message M.

The complexity of the attack is total complexity of finding a slid pair and reversing one of the final states. Attacker calculates 2^{64} reverse blank rounds to find a slid pair. Then computes 2^{80}, again, reverse blank rounds to recover the actual values of 10 unknown bytes. The extension of the message will be negligible. Therefore the total complexity of the slide attack on Grindahl is $O(2^{64} + 2^{80}) = O(2^{80})$.

4.11 Rebound Attack

Rebound attack [79] is a differential collision method for block cipher based hash functions. The aim of the attack is, given two messages M_1 and M_2 such that $M_1 \oplus M_2 = \Delta$, to cancel the difference Δ after some number of iterations or some rounds of encryption in the compression function.

Attack divides the block cipher of the compression function into three subciphers as $E_{fw} \circ E_{in} \circ E_{bw}$ and proceeds in two phases: the inbound phase and the outbound phase. In the inbound phase attacker tries to find a match in the middle using the degrees of freedom in E_{in} subcipher. Then going backwards and forwards through the matching pairs, tries to get a zero difference at the end, which is produced by the Δ difference. This phase is called the outbound phase and processed through E_{fw} and E_{bw}. The inbound phase produces starting points(states) for outbound phase, therefore, inbound phase should be repeated as many times as needed according to the probability of the outbound phase.

Despite the attack can be applied to any block cipher based hash function, it has been applied to, only, hash functions, like Whirlpool[80], Maelstrom [81], and Grøstl [82], which use Rijndael [83] building blocks. In the rest of this section, rebound attack will be described on Whirlpool hash function.

Whirlpool is a 512-bit block cipher based iterative hash function which is a Miyaguchi-Preneel construction designed by Rijmen and Barreto. The block cipher W and the key scheduling algorithm are the same up to the constants and uses transformations which are similar to Rijndael round functions with 64 bytes state. One round consists of four transformations SubBytes, ShiftColumn, MixRow and AddConstant and both W and the key scheduling algorithm is 10 rounds. The detailed description of Whirlpool can be found in [80].

The attack is applied to 4 rounds Whirlpool which assumes W and key scheduling in the compression function is of 4 rounds. For simplicity the attack described below searches for a collision after a single iteration.

Since the attack is a differential attack, one should search for characteristics with small number of differentials in order to proceed efficiently (with low complexity). For Whirlpool the optimal characteristic is of the form $1 \rightarrow 8 \rightarrow 64 \rightarrow 8 \rightarrow 1$ which means a 1-byte difference

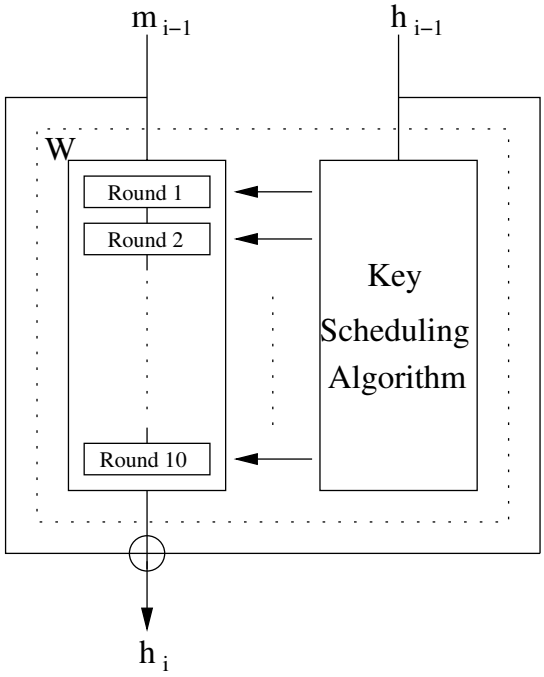

Figure 4.21: Compression Function of Whirlpool Hash Function

diffuses to 8 and 64 bytes after 1^{st} and 2^{nd} rounds respectively. Then these differences cancel out each other to a 1-byte difference again. If 1-byte differences in the beginning and at the end of the characteristics are equal, then after feed forward operation one gets a collision.

First the block cipher W is divided into three subciphers $W = W_{fw} \circ W_{in} \circ W_{bw}$ as

$$\begin{aligned} W_{bw} &= SC \circ SB \circ AK \circ MR \circ SC \circ SB \\ W_{in} &= MR \circ SC \circ SB \circ AK \circ MR \\ W_{fw} &= AK \circ MR \circ SC \circ AB \circ AK \end{aligned}$$

where SB, SC, MR and AK are synonyms for transformations SubBytes, ShiftColumn, MixRow and AddConstant respectively.

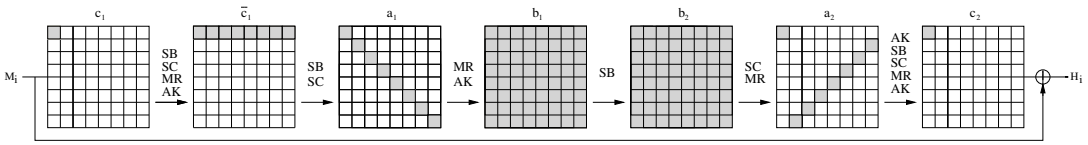

Figure 4.22: Differential Characteristic for Rebound Attack on Whirlpool

The attacker produces two state differences a_1 and a_2 with 8-byte non-zero differences at

specific locations as shown in the *Figure 4.22*. Then computes $b_1 = AK(MR(a_1))$ and $b_2 = SC^{-1}(MR^{-1}(a_2))$ which have 64-byte non-zero differences each. If b_1-b_2 is a possible[3] input difference - output difference pair for SubByte transformation, she can use a_1, a_2 and the actual values corresponding to these differences in the outbound phase. Otherwise, she should restart the attack by choosing new state differences. To check if b_1 - b_2 is a possible pair, one can prepare a look-up table before starting the attack and store the possible input-output differentials. For the s-box of Whirlpool, there are 25881 possible pairs having values 2,4,6 and 8, out of all 65536 pairs. Therefore, the probability that an input difference, in a single byte, is a possible pair with the output difference in the corresponding byte is $\frac{25881}{65536} \simeq 0,4$. Hence, a full state of 64-bytes can be a possible pair with a given output difference state with probability $(0,4)^{64} = 2^{-85,7}$. Note that each possible single-byte input-output difference pair (i, p) will produce a 2,4,6 or 8 possible byte pairs B_1, B_2 with $B_1 \oplus B_2 = i$ and $SB(B_1) \oplus SB(B_2) = p$. The distribution of the differentials for s-box of Whirlpool is given in [79]. According to this distribution one can produce $2^{79,7}$ "possible" state values. So the attacker gets $2^{79,7}$ matching states with a complexity $2^{85,7}$. Now she can use these state values for the outbound phase of the attack.

Going backwards from a_1 and forwards from a_2, the 8-byte differences are preserved with no cost. The probability of, both forwards and backwards, MixRow transformation where 8-byte difference reduces to a single-byte difference is 2^{-56}.[4] Therefore to get a single byte difference in the beginning and at the end of the characteristic attacker needs $2^{2 \cdot 52} = 2^{112}$ matching states. Since one match in the inbound phase produces $2^{79,7}$ states, in order to have one byte differences in the beginning and at the end, inbound phase should be repeated $2^{112-79,7} = 2^{32,3}$ times which costs $2^{89,7+32,3} = 2^{118}$. Moreover, with 2^{-8} probability these two differences will be the same and after the feed forward operation the difference at the end will vanish. So, attacker will get a collision after one block iteration of 4-round Whirlpool with messages M_1 and M_2 such that $M_1 \oplus M_2 = c_1$, after 2^{126} trials. Note that the initial chaining values are the same for both messages, therefore, the keys and the keys derived from the key schedule will be the same and will not affect the characteristic. Also, since there are 16 bytes that can be selected randomly for a_1 and a_2 attacker can have 2^{128} starting points which is enough for an attack of complexity 2^{126}.

[3] Possible input-output pairs are those which have non-zero values, in Whirlpool case 2,4,6 and 8, in the difference distribution table

[4] The probability is determined by searching over all possible differences in [79].

The 4-round attack can be extended to 4.5-rounds by adding the SubBytes and ShiftColumn transformations with no cost.

Moreover in [79], the attack is applied to Grøstl and Maelstrom besides Whirlpool, and can be extended to 5.5-rounds semi-free-start attack and 7.5-rounds(8.5 for Maelstrom) free-start attacks.

CHAPTER 5

CONCLUSION

Cryptographic hash functions have been used in many daily life applications. However, the knowledge about these tools are very little. Cryptographic community have been dealing intensively with the hash functions for the past few years. $SHA-3$ competition fuels this attention. After the competition, whichever algorithm is chosen, it is certain that community will have learned much about designing and analyzing hash functions.

This thesis is serving a as an extensive survey on hash functions. Starting from very basic definitions and properties, it covers design parameters and cryptanalytic attacks.

The thesis can be divided into three main parts. First part is an introduction to hash functions with applications. Second part, covering Chapter 2 and Chapter 3, is the design part. In tis part compression functions and construction methods are dealt with. Finally, third part is the analysis part. This part includes 11 cryptanalytic methods and precautions that can be taken against these attacks.

REFERENCES

[1] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *EUROCRYPT*, pages 19–35, 2005.

[2] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In *EUROCRYPT*, pages 1–18, 2005.

[3] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *CRYPTO*, pages 17–36, 2005.

[4] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision search Attacks on SHA-0. In *CRYPTO*, pages 1–16, 2005.

[5] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[6] R. Anderson. The Classification of Hash Functions. In *IMA Conference in Cryptography and Coding*, pages 83–93, 1993.

[7] C.H. Meyer J. Oseas S.M. Matyas. Generating Strong One-Way Functions with Cryptographic Algorithm. In *IBM Techn. Disclosure Bull., Vol. 27, No. 10A*, pages 5658–5659, 1985.

[8] S. Miyaguchi, M. Iwata, and K. Ohta. New 128-Bit Hash Function. In *Proceeding of* 4^{th} *International Joint Workshop on Computer Communications*, pages 279–288, 1989.

[9] B. Preneel, R. Govaerts, and J. Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. pages 368–378, 1993.

[10] D. Coppersmith M. Hyden S. Matayas C. Meyer J. Oseas S. Pilpel B. Brachtl and M. Schiling. Data Authentication Using Modification Detection Codes Based on a Public One-Way Encryption Function. In *U. S. Patent Number 4,908,861*, 1990.

[11] J. Black, P. Rogaway, and T. Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions From PGV. In *In Advances in Cryptology - CRYPTO 2002*, pages 320–335. Springer-Verlag, 2002.

[12] Martijn Stam. Blockcipher Based Hashing Revisited. Cryptology ePrint Archive, Report http://eprint.iacr.org/2008/071.pdf, 2008.

[13] Lars Knudsen and Bart Preneel. Hash Functions Based on Block Ciphers and Quaternary Codes. In *Advances in Cryptology ASIACRYPT*, pages 77–90, 1996.

[14] Joan Daemen and Craig Clapp. Fast Hashing and Stream Encryption with Panama,. In *Fast Software Encryption 1998*, pages 60–74. Springer.

[15] Donghoon Chang, Kishan Chand Gupta, and Mridul Nandi. Rc4-hash: A New Hash Function Based on RC4. In *INDOCRYPT*, pages 80–94, 2006.

[16] Vincent Rijmen, Bart Van Rompay, Bart Preneel, and Joos Vandewalle. Producing Collisions for Panama. In *Fast Software Encryption*, pages 37–51. Springer, 2002.

[17] Joan Daemen and Gilles Van Assche. Producing Collisions for Panama, Instantaneously. In *Fast Software Encryption*, pages 1–18. Springer, 2007.

[18] Guido Bertoni, Joan Daemen, and Gilles Van Assche. Radiogatun, A Belt-and-Mill Hash Function. In *NIST - Second Cryptographic Hash Workshop*, 2006.

[19] Ivica Nikolić, Alex Biryukov, and Dmitry Khovratovich. Hash family LUX - Algorithm Specifications and Supporting Documentation. Submission to NIST, 2008.

[20] Gregory G. Rose. Design and Primitive Specification for Boole. Submission to NIST, 2008.

[21] Burt Kaliski. The MD2 Message-Digest Algorithm. http://www.ietf.org/rfc/rfc1319, 1990.

[22] Ron Rivest. The MD4 Message-Digest Algorithm. http://tools.ietf.org/rfc/rfc1320, 1990.

[23] Hans Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[24] Hans Dobbertin. Cryptanalysis of MD4. In *Fast Software Encryption*, volume 1039 of *LNCS*, pages 53–69, 1996.

[25] Ron Rivest. The MD5 Message-Digest Algorithm. http://tools.ietf.org/rfc/rfc1321, 1991.

[26] National Institute of Standards and Technology. FIPS PUB 180. http://security.isu.edu/pdf/fips180.pdf, 1993.

[27] National Institute of Standards and Technology. FIPS PUB 180-1. http://www.itl.nist.gov/fipspubs/fip180-1.htm, 1995.

[28] D. W. Davies and W. L. Price. The Application of Digital Signatures Based on Public-Key Cryptosystems. In *Proc. Fifth Intl. Computer Communications Conference*, pages 525–530, 1980.

[29] ITU-T X.500. The Directory - Overview of Concepts Models and Services., 1988.

[30] Robert R. Jueneman. Analysis of Certain Aspects of Output Feedback Mode. In *CRYPTO*, pages 99–127, 1982.

[31] Robert R. Jueneman, Stephen M. Matyas, and Carl H. Meyer. Message Authentication with Manipulation Detection Code. In *IEEE Symposium on Security and Privacy*, pages 33–54, 1983.

[32] Robert R. Jueneman. A High Speed Manipulation Detection Code. In *CRYPTO*, pages 327–346, 1986.

[33] Ivan Damgård. A Design Principle for Hash Functions. In *CRYPTO*, pages 416–427, 1989.

[34] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental Cryptography: The Case of Hashing and Signing. In *CRYPTO 94*, pages 216–233, 1994.

[35] J.K. Gibson. Discrete Logarithm Hash Function That is Collision Free and One-Way. In *IEE Proceedings-E*, volume 138, pages 407–410, November 1991.

[36] Paul Camion and Jacques Patarin. The Knapsack Hash Function Proposed at Crypto'89 Can Be Broken. In *EUROCRYPT*, pages 39–53, 1991.

[37] Jacques Patarin. Collisions and Inversions for Damgård's Whole Hash Function. In *ASIACRYPT*, pages 307–321, 1994.

[38] M. Ajtai. Generating Hard Instances of Lattice Problems (Extended Abstract). In *In Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 99–108, 1996.

[39] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Collision-Free Hashing from Lattice Problems. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(042), 1996.

[40] Gilles Zémor. Hash Functions and Cayley Graphs. *Des. Codes Cryptography*, 4(4):381–394, 1994.

[41] Jean-Pierre Tillich and Gilles Zémor. Hashing with SL_2. In *CRYPTO*, pages 40–49, 1994.

[42] Willi Geiselmann. A Note on the Hash Function of Tillich and Zémor. In *IMA Conf.*, pages 257–263, 1995.

[43] Chris Charnes and Josef Pieprzyk. Attacking the SL_2 Hashing Scheme. In *ASIACRYPT*, pages 322–330, 1994.

[44] Henk C. A. van Tilborg, editor. *Encyclopedia of Cryptography and Security*. Springer, 2005.

[45] Joan Daemen, René Govaerts, and Joos Vandewalle. A Framework for the Design of One-Way Hash Functions Including Cryptanalysis of Damgård's One-Way Function Based on a Cellular Automaton. In *ASIACRYPT*, pages 82–96, 1991.

[46] Joan Daemen, René Govaerts, and Joos Vandewalle. A Hardware Design Model for Cryptographic Algorithms. In *ESORICS*, pages 419–434, 1992.

[47] Donghoon Chang. Preimage Attacks on Cellhash, Subhash and Strengthened Versions of Cellhash and Subhash. http://eprint.iacr.org/2006/412, 2006.

[48] B. Banieqbal and S. Hilditch. The Random Matrix Hashing Algorithm. Technical Reports UMCS-90-9-1, Department of Computer Science, University of Manchester, 1990.

[49] Sami Harari. Non-Linear Non-Commutative Functions for Data Integrity. In *EUROCRYPT*, pages 25–32, 1984.

[50] Ralph C. Merkle. One Way Hash Functions and DES. In *CRYPTO*, pages 428–446, 1989.

[51] Douglas R. Stinson. *Cryptography: Theory and Practice, Second Edition*. Chapman & Hall/CRC, 2002.

[52] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions: HAIFA. In *In Proceedings of Second NIST Cryptographic Hash Workshop,*.

[53] Richard D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, 1999.

[54] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In *CRYPTO 2004*, pages 306–316. Springer.

[55] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In *EUROCRYPT 2005*, pages 474–490. Springer.

[56] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions, 2007.

[57] Stefan Lucks. Design Principles for Iterated Hash Functions, 2004.

[58] Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In *ASIACRYPT 2005*, pages 474–494. Springer.

[59] Jean-Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to Construct a Hash Function. In *CRYPTO 2005*, pages 430–448. Springer-Verlag.

[60] Mihir Bellare and Thomas Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In *ASIACRYPT 2006*, pages 299–314. Springer-Verlag.

[61] Shai Halevi and Hugo Krawczyk. The RMX Transform and Digital Signatures.

[62] Shai Halevi and Hugo Krawczyk. Strengthening Digital Signatures via Randomized Hashing. In *CRYPTO 2006*, pages 41–59. Springer.

[63] Prevee Gauravaram. *Cryptographic Hash Functions: Cryptanalysis, Design and Applications*. PhD thesis, 2007.

[64] William Speirs. *Dynamic Cryptographic Hash Functions*. PhD thesis, 2007.

[65] Josef Pieprzyk and Babak Sadeghiyan. *Design of hashing Algorithms*. Springer-Verlag, 1993.

[66] K. Nishimura and M. Sibuya. Probability to Meet in the Middle. *J. Cryptology*, 2(1):13–22, 1990.

[67] Don Coppersmith. Another Birthday Attack. In *CRYPTO*, pages 14–17, 1985.

[68] Marc Girault, Robert Cohen, and Mireille Campana. A Generalized Birthday Attack. In *EUROCRYPT*, pages 129–156, 1988.

[69] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons Inc, 2003.

[70] Don Johnson. Hash Function Lifesycles and Future Resiliency. Technical Reports National Institute of Standards and Technology NIST October 2005, This paper is available at http://csrc.nist.gov/groups/ST/hash/documents/Johnson_LifecyclesResiliency.pdf. Last access date: 19^{th} of June 2009.

[71] M. Nandi and D. R. Stinson. Multicollision Attacks on a Class of Hash Functions, 2005.

[72] Jonathan J. Hoch and Adi Shamir. Breaking the ICE - Finding Multicollisions in Iterated Concatenated and Expanded (ice) Hash Functions. In *Fast Software Encryption*, pages 179–194, 2006.

[73] Ralph C. Merkle. A Fast Software One-Way Hash Function. *J. Cryptology*, 3(1):43–58, 1990.

[74] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In *EUROCRYPT*, pages 183–200, 2006.

[75] Orr Dunkelman and Bart Preneel. Generalizing the Herding Attack to Concatenated Hashing Schemes. In *ECRYPT Hash Workshop*, 2007.

[76] Alex Biryukov and David Wagner. Slide Attacks. In *Fast Software Encryption 99*, pages 245–259, 1999.

[77] Michael Gorski, Stefan Lucks, and Thomas Peyrin. Slide Attacks on a Class of Hash Functions. In *ASIACRYPT*, pages 143–160, 2008.

[78] Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In *Fast Software Encryption*, pages 39–57, 2007.

[79] Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren Steffen Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In *Fast Software Encryption*, pages 265–281, 2009.

[80] Paulo S. L. M. Barreto and Vincent Rijmen. The Whirlpool Hashing Function. Submitted to NESSIE, September 2000 Revised May 2003. Available at urlhttp://www.larc.usp.br/˜pbarreto/WhirlpoolPage.html Last access 9^{th} June 2009.

[81] Decio Gazzoni Filho, Paulo S.L.M. Barreto, and Vincent Rijmen. The Maelstrom-0 Hash Function. In *SBSeg 2006*, 2006.

[82] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 Candidate. Submission to NIST, 2008.

[83] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.